NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PERFORMANCE EVALUATION OF THE JRS
AUTOMATIC MICROCODE GENERATING SYSTEM

by

Terry J. Newton

June 1985

Thesis Advisor: Alan A. Ross

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Performance Evaluation of the JRS Automatic Micrcode Generating System	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis, June 1985
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Terry J. Newton	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943	12. REPORT DATE: June 1985
	13. NUMBER OF PAGES: 120
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

microprogramming, microcode, computer performance evaluation, automatic microcode generation, microoperations, microinstruction, machine dependence, microcode compaction, code optimization, benchmark

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The purpose of this thesis was to evaluate the performance of microcode automatically generated from a high level language. The performance of the generated microcode was compared to the performance of Fortran code on the VAX 11/780 to see if any increase in throughput could be attained by using the microcoded version.

The factors affecting the automatic generation of microcode: compaction, optimization, cost, and machine (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ABSTRACT (Continued)

independence are discussed. This is followed by a definition of the testing areas, description of the tests, and a description of the performance evaluation methods.
The tests showed that the automatically generated microcode does not always out perform Fortran. In general, the Fortran code was better for mathematical calculations while the automatically generated microcode was better for bit manipulation and sorting/ searching type applications.

Performance Evaluation of the JRS Automatic Microcode Generating System

by

Terry J. Newton
Captain, United States Air Force
B.S., United States Air Force Academy, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

ABSTRACT

The purpose of this thesis was to evaluate the performance of microcode automatically generated from a high level language. The performance of the generated microcode was compared to the performance of Fortran code on the VAX 11/780 to see if any increase in throughput could be attained by using the microcoded version.

The factors affecting the automatic generation of microcode: compaction, optimization, cost, and machine independence are discussed. This is followed by a definition of the testing areas, description of the tests, and a description of the performance evaluation methods.

The tests showed that the automatically generated microcode does not always out perform Fortran. In general, the Fortran code was better for mathematical calculations while the automatically generated microcode was better for bit manipulation and sorting/searching type applications.

TABLE OF CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

Thanks are due to Mr. Bob Sheraga for his assistance in explaining the VAX 11/780 microcode and micromachine. He also performed the compaction of the HLL microcode. Without his expertise and assistance, the compaction would not have been accomplished and that part of the testing would have been lost.

Lt. Col. Ross was very supportive during the entire project. His guidance and support, along with his probing questions, were very important in finishing the paper.

Most importantly, I thank my family, Karen, Chad, and Lisa, for their support and understanding during the period while I was writing this paper.

I. INTRODUCTION

One of the problems currently being studied by the Naval Research Laboratory (NRL) involves the processing of frequent and complex messages from satellites. The processing of these messages requires a high percentage of bit manipulations which uses a large amount of central processing unit (CPU) time. The currently available computers do not have sufficient capability to perform this processing in a timely manner. There are several options available to the NRL for improving the situation. One option is the use of a very fast computer, however, the cost of such a computer is very high. The purpose of this project is to evaluate another less costly option using an automatic microcode generating system (AMGS).

JRS Research Laboratories Inc. has developed an AMGS which generates microcode for the writeable control store (WCS) on the VAX 11/780. The JRS AMGS was developed to provide a low cost technique for algorithm implementation which provides the performance of microcode, yet does not require detailed machine level microprogramming. The JRS AMGS is a software package that generates microcode from a high level language (HLL), thereby eliminating the need for the programmer to be concerned with the details of microcode. The user, therefore, need not understand

microcode programming and may apply the principles of software engineering through the use of an HLL. Figure 1-1 [Ref. 1: p. 559] shows the steps involved in generating microcode from the HLL using the AMGS. It is important to note where the AMGS is machine independent and where it is machine dependent. This will be important in later discussions of the system.

Since the target machine of the JRS AMGS, the VAX 11/780, is a horizontally microprogrammed processor, it is capable of executing a number of operations simultaneously. This is the key ingredient to improving the speed and efficiency of the executable code because several microoperations may be executed concurrently. [Ref. 1: pp. 558-559] By applying the JRS AMGS to the data manipulation requirements of the satellite communication problem, a reduction in required CPU time should be achieved.

Since the current method used by NRL for implementing the algorithms is to write them in Fortran, this project will compare the execution speed attained using the AMGS to the execution speed attained using Fortran code. The results of this comparison will provide an understanding of the type of algorithms that are suitable for implementation via the JRS AMGS, the performance improvements to these algorithms, and the costs of using this implementation technique. This study is based on two aspects of computer science: microprogramming and computer performance

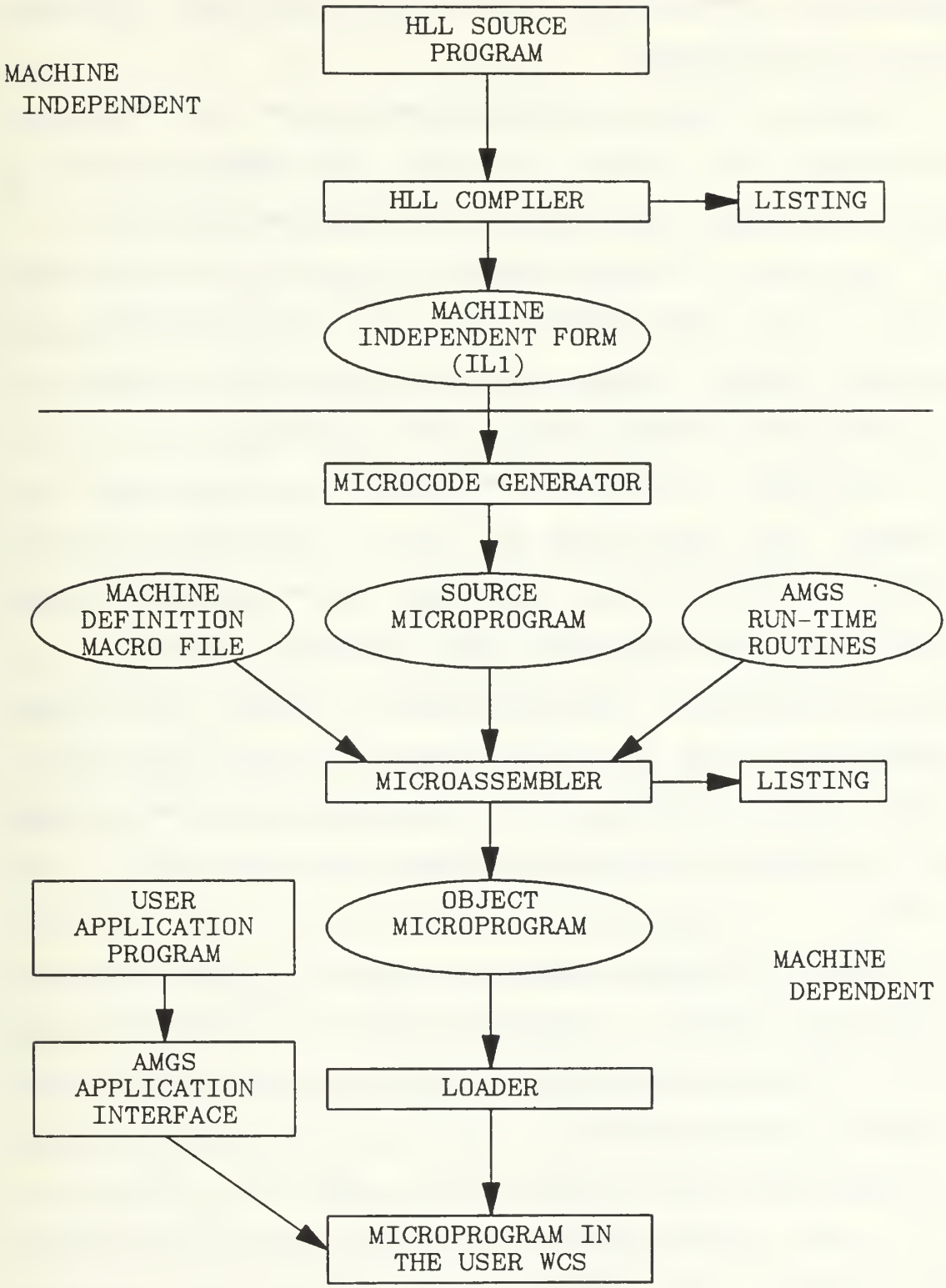

Figure 1-1: AMGS Operation

evaluation. Before the study can be described, these two areas must be defined.

Wilkes defined microprogramming as a method of implementing the control function of a computer. [Ref. 2: p. 59] The major advantages of microprogramming are:

1) Low Cost: Microprogramming allows large instruction sets to be implemented at a low cost because of the simple design process. Developing a hardwired design of an equivalent system would be very expensive.

2) Flexibility: With microprogramming, it is possible to change the instruction set or to introduce a new set after implementation. This may allow a computer to be useful for many more years than originally planned.

3) Simplicity: Microprogramming allows for simpler development due to the decrease in internal circuitry. This simpler design facilitates maintenance and reduces the problems associated with upgrading the design in the field.

4) Speed: Although microprogramming is slower than some hardwired designs, a microprogrammed implementation will run faster on most algorithms than an equivalent machine language implementation. This is due to the machine language fetch and decode overhead. [Ref. 3: p. 5]

A major disadvantage of microprogramming is the memory delay penalty for fetching each microinstruction from the control store. This fetch penalty can result in slow

execution times if not dealt with properly, but the problem can be made less significant by providing an overlap between the fetch and the execute portions of microinstructions. [Ref. 3: p. 5]

Since this project is concerned with comparing Fortran code with microcode, it is important to review the tradeoffs between using Fortran (or some other HLL) and microcode. Programming in microcode is very tedious and complex because the programmer must deal with the details of the machine. However, it is this complexity of microcode that can, through proper programming, lead to a speed advantage. On the other hand, Fortran and similar HLLs are not nearly as complex because the details of the machine are handled by the compiler. The slower execution speed for HLL's results from both the generalization required in the code generation portion of the compiler and the instruction fetch decode penalty described in the explanation of the microcode speed advantage.

Microprogramming, in its present state, may be used to provide efficient implementations of the control function on computers. While not providing the fastest execution speed possible, microcoding will provide a given level of throughput at a cheaper price than is otherwise possible. In addition, the speed of microcode has recently improved because of the development of fast, inexpensive semiconductor memories. These are the two main reasons to

suggest that the AMGS can give a performance advantage over Fortran source code.

Since this study involves the evaluation of the performance of microcode, it is important to review the relevant performance evaluation techniques, methods, and problems. The performance evaluation in this study is a comparison between different implementations of the same algorithm. The classic application of performance evaluations is on operating systems to determine how to improve the system. To achieve the comparison, the evaluators must define a benchmark which represents the type of workload that occurs on that system. Defining this workload properly and accurately is very important or the results will be invalid. In the case of this study, a major consideration is the definition of the algorithms to be implemented and compared.

One option when picking the algorithms is to choose a very specific application area and test only within that area. Another option is to attempt to test the entire realm of possible applications which would take many different algorithms. In Chapter Three, the application areas of interest will be defined and the subsets of these areas to be tested will be identified. The tests will be as comprehensive as possible and will cover as large an area as possible, however, exhaustive testing of the entire realm of applications is not possible.

The evaluation technique will consist of implementing the algorithm in both Fortran and in the AMGS HLL. Both the Fortran and the HLL codes will be executed, timed, and the execution times will be compared to determine the change in performance with the HLL microcode version. Since the algorithms are grouped according to application, it is possible to determine which applications have increased throughput from use of the AMGS.

Several contributing factors must be considered during this performance evaluation. The effect of using an HLL instead of assembly language or direct microprogramming to implement the microcode version is significant because of the costs involved in using each method. Similar costs are associated with all HLLs, be it the JRS HLL and its associated compiler or Fortran and its resultant translation. Likewise looking at the more primitive languages of assembly code and microcode, the costs of programming in both languages are very similar. However, not so obvious is the tradeoff involved in choosing one type of language (high level versus low level) over another. The specific compilation techniques may also have an effect on the efficiency of the product and must be considered. The microcode compaction method used will certainly affect how fast the microcode executes. A performance evaluation must analyze many factors, both individually and combined, to produce valid results.

Chapter Two is a discussion of the issues of AMGS design, compiler technique, code generation, and code optimization. The purpose of this discussion is to assess the effect each item has on the entire system so that during the evaluation of the AMGS these factors can be properly analyzed.

Chapter Three describes how each program was generated. Because the AMGS is being evaluated for all applications, this chapter defines the basic areas that are tested in the project. After the basic areas are defined, the specific tests developed to cover these areas are explained and the information to be gained from each test is outlined.

Chapter Four explains the mechanics of the testing including the timing mechanism and the effects of language features on the tests. An analysis of possible sources of errors is also included here to explain the validity of the results.

Chapter Five compares the data from the tests and analyzes the results. A step-by-step explanation of the testing is enumerated to insure a proper understanding of why certain tests were accomplished. The last chapter summarizes the results and makes deductions and recommendations for further research in this area.

II. BACKGROUND

With the many advantages of microprogrammed computers there is no apparent reason why microprogramming should not be used for most applications. Low cost, fast execution time, and simplicity of design sound like exactly what a computer designer desires. There is, however, the problem of developing the microprograms (commonly called firmware) for the computer in a reasonable amount of time and with reasonable cost. Developing firmware has been a costly, error prone, and slow process because it has been done manually and because of the details that must be handled by the microprogrammer.

The obvious answer is to eliminate the use of low level languages and place the microprogrammer into the world of high level languages. That is the intent of the AMGS. However, along with the advantages of an HLL come problems and considerations that cannot be ignored. This chapter will explore the many considerations of the AMGS and discuss their impact on the JRS AMGS. The following topics are considered to be the most relevant and will be discussed in depth in this chapter: 1) High Level Languages and Microprogramming, 2) Machine Independence, 3) Compaction and Optimization, 4) JRS AMGS Limitations, and 5) Performance Evaluation Methodology.

A. HIGH LEVEL LANGUAGES AND MICROPROGRAMMING

Higher level languages are designed to simplify programming by isolating the programmer from the details of the machine and placing him at a higher level of abstraction. An AMGS removes the programmer from the details of microprogramming and allows the programmer to write the code in an HLL. Writing a program in an HLL takes much less time than writing the same program in microcode because the programmer must deal with fewer details. There are many studies that have shown the advantages of using HLLs instead of assemblycode. One such study claims that a programmer produces a set number of lines of code per day, independent of the type of code. Since one line of HLL code will produce many lines of microcode, it is logical to opt for the HLL if all other factors are equal. [Ref. 4: p. 145]

However, all other factors are not equal. Since the HLL used for generating the microcode is a special purpose language, any program written to use this system must be translated from another language before it can be used. In this particular case, the time required to program in the HLL provided by JRS must be considered. Of course the time required to translate the algorithm to the HLL should be much less than would be required to translate the algorithm into assembly language code or into microcode. Since the JRS HLL is a language heavily influenced by the block

structure of Algol, Fortran, and Pascal, any algorithm written in a block structured language should be easy to translate to the JRS HLL.

Software engineering literature provides many reasons for using HLLs. One such reason is that HLLs provide security not possible using microcode or assemblycode. Forced typing of variables is one example of the security provided by HLLs. High level languages also provide features such as subprograms which are an advantage because they assist the user in subdividing the program into logical units. These logical units make the problem easier to understand and handle.

The chief advantage of an HLL is the ease of program maintenance which results in lower life cycle costs for a program. Program changes can be very expensive if the programmer must read and understand low level code, with or without good documentation. Through use of an HLL, program changes can be made much more quickly and simply, with reduced costs.

The advantages of HLLs are all 'nice' for the programmer, but it is important to consider the side-effects of HLL usage. If the advantages of an HLL detract from the advantages of microprogramming (i.e.- simplicity, cost, flexibility, speed) then using the AMGS may not be justified. On the other hand, if the AMGS eliminates or minimizes other problems of microcode, then

the AMGS will become even more desirable. One such undesirable property of microcode is its machine dependence. In the next section we will look at the effect of using an HLL on the machine dependence of the resulting microcode.

B. MACHINE INDEPENDENCE

Machine independence is a major concern during the development of an AMGS because of the desire to make firmware portable. The AMGS is a tool used to help achieve the goals of machine independence and portability. Machine independence and portability are desirable characteristics for any computer language because if the code may be used on more than one computer, the overall firmware development costs will be lower. If every different target machine must have its own version of the algorithm written specifically for it, then the cost of program development will be a function of the number of target machines. A much more desirable method is to write one program that may be used on every machine resulting in only one program being developed.

A machine dependent language is "a language in which all operations and data elements defined in the language have a direct mapping to a resource of the target machine." [Ref. 5: p. 194] The actual microcode is such a language because it specifically addresses the available registers

of a machine. To remain machine independent and avoid the problems of machine dependence, a language must avoid the details of the target machine and remain general enough to not become tied to any specific instruction set. This can be accomplished by defining an overall class of machines and then writing the language to fit into that definition. Such a definition includes such items as the minimum number of registers, the minimum stack size, and other hardware related items. Capitalizing on the similarities and avoiding the differences of the machines in the class simplifies this task. Any item that is not common to all machines in the class must not be included in the definition because it can not be supported by all machines in the class.

The AMGS supports machine independence and portability of microcode by providing an intermediate language and an HLL that avoids the direct mapping to the machine resources. Since these two components of the AMGS are machine independent, a user may write a program in the AMGS HLL and then use the code on different target machines. The major problem is making the transition from the machine independent intermediate language to the target machine's microcode. To achieve this, each machine requires a separate code generator to translate the intermediate language to the microcode level plus a compactor to compact the resulting microcode. This is not a trivial step and

there is currently considerable research being conducted on microarchitecture description techniques that will assist in making this step easier. Geiser has introduced a description methodology that covers four basic areas:

1) Microinstruction description: includes the format of the microinstruction, fields used in the microinstruction, and possible values in each field.

2) Element descriptions: describes and names elements of the machine hardware including memory, registers, etc.

3) Microoperation usage rules: a set of rules for constructing valid microoperations.

4) Microengine behavioral rules: specifies interactions between the microoperations. [Ref. 6: pp. 517 - 521]

By using this technique it is possible to describe the target machine in a standardized format so that the writing of the machine dependent code generator is much easier.

Of course the description methodology does not eliminate the problem. The main purpose of a description methodology is to reduce the work required to port the language to another machine by maximizing the common features of the different machine dependent languages. This may eliminate desirable machine dependent features but it does permit a 'nearly machine independent' language. The assumption is if you cannot be totally independent then be as independent as possible. [Ref. 5: p. 195]

True machine independence has not been achieved in this AMGS and probably will not be achieved in the near future, however the microarchitecture description methodology is an attempt at reducing the portability problem. By providing a systematic description of microarchitectures, the description methodology reduces the amount of work required to move a system to another comparable machine. The AMGS is providing a step toward an ultimate goal of machine independence that may never be achieved. However, the AMGS has helped to define and simplify some of the steps involved in making microcode generation less machine dependent.

C. COMPACTION AND OPTIMIZATION

Before reviewing the current compaction techniques it is important to understand the difference between compaction and optimization. Microcode compaction will reduce the space required to store a program but does not guarantee a reduction in the speed of execution. Optimization, on the other hand, results in a reduction in execution speed but does not guarantee that any code compaction will occur. Sometimes execution time will decrease when the code is compacted, but the reduction of execution time is not guaranteed, in fact execution time can in some cases increase. The only conclusion that can be drawn is that successful compaction guarantees fewer

total instructions and may lead to faster or possibly slower execution speed.

Most HLL compilers do include an optimization step, however, the present version of the JRS HLL compiler does not. There are two reasons for this. One reason is that excessive optimizations prior to microcode generation can make error correction very difficult because of the movement of the microoperations. Secondly, since this was the first production version of an AMGS, some of the more difficult problems were not handled. Optimizing the compiler without excessively affecting error correction is one of the more difficult problems. The AMGS does as a whole include a number of optimization steps designed to produce more efficient microprograms. An example of such a step is the use of registers to hold array offset addresses to help reduce memory fetch delay. Even though none of the common compiler optimization techniques are used in this system, it is important to discuss them here to understand the effect they could have on microcode compaction.

Gries gives a good explanation of the four main compiler optimization techniques that are applicable to almost any algebraic programming language such as Fortran, Pascal, Algol, PL/I, etc. The four methods are:

1) Folding: for any operator whose operands are known at compile time, perform the applicable operation at compile time rather than at execution time.

2) Eliminating redundant operations: mainly factoring out common subexpressions.

3) Moving operations out of loops if their operands do not change within the loop.

4) Reducing the number of multiplications in loops: effectively changing the multiplications to additions. [Ref. 7: pp. 376 - 377]

A system may use these techniques to attain whatever level of optimization is desired, however there is a tradeoff between the level of optimization and the time required to perform the compilation. Also as mentioned above, extensive optimization will result in radically altering the sequence of operations and therefore make debugging very difficult. [Ref. 7: p. 376]

Even though optimization is important, there has been very little work done on optimization of microcode. Almost all of the work done on microcode has been in the field of compaction because optimization of microcode is very difficult to do systematically and is not well understood. Most microcode compaction research has been justified by the assumption that execution time will decrease when the code is compacted. It is important to keep this assumption in mind when discussing compaction because the results of the compaction are not guaranteed to reduce execution time and will certainly not optimally reduce execution time. However, compaction is the only automated method for

improving microcode that is currently available for practical use.

It is important to remember the assumption that the target machine will be horizontally microprogrammable, meaning that more than one operation may be executed during any microinstruction. If the target machine is not horizontally microprogammable, then only one microoperation may occur during any microinstruction (or machine cycle) and compaction is not possible. There are two classes of microcode compaction for horizontally microprogrammable computers, local and global, and a discussion of the compaction techniques from both classes will follow. JRS does not do any code compaction in this version of the AMGS. However, by reviewing the many methods of compaction available it will be evident which methods are the most promising for future improvements.

Local compaction of microcode is concerned with the reduction of the number of microinstructions in a straight-line code (SLC) segment of a microprogram. An SLC segment is any sequence of microinstructions that begins either at the start of the program or after a branch statement and ends either at the end of the program or at a branch statement. Only one entrance and one exit is allowed in any SLC segment. Local compaction is simply an attempt at reducing the number of microinstructions in each SLC segment by combining instructions or eliminating

duplicated instructions. The most promising and popular versions are first-come first-serve, critical path, branch and bound, and list scheduling.

First-come first-serve is probably the simplest form of local compaction possible. Each microoperation is considered only once, in source code order, and in the SLC segment that it exists. Each microoperation is moved as far forward in its segment as possible. If it can be combined with a previous operation without causing a conflict, then it will be combined. Once a microoperation has been checked and combined or not combined, it will never be considered again. This results in fast compaction but the resulting microcode is not optimally compacted. [Ref. 8: p. 415]

Critical path algorithms compact microcode in each SLC segment by identifying microoperations "that cannot be delayed without increasing the number of microinstructions needed for the microprogram." [Ref. 8: p. 415] This is accomplished by first identifying the longest paths in the data dependency graph. Each of the longest paths is called a critical path and shortening the path will result in a more compact program. Each microoperation in each critical path is checked to see it if can be moved forward and combined with another microoperation. If it can be moved forward, the critical path will be shortened and the result is a more compact program. If any microoperation in any of

the critical paths is delayed (not forwarded as much as possible), then the trailing microoperations will be delayed, which will result in more microinstructions than are actually needed and less compact microcode. [Ref. 8: p. 422] Once again the results are not optimal and the time required to do the compaction is a polynomial function of the number of microoperations which are considered in each SLC segment.

Branch and bound algorithms can guarantee optimality in storage space required for the microprogram. Remember that this says nothing about the execution time of the program. The method depends upon searching a tree structure exhaustively, looking for the optimal ordering. This method may produce optimal compaction, but the time required is an exponential function of the number of microoperations in the microprogram, making the method very expensive. There are variations to the branch and bound algorithms that are not so expensive. One such variation involves pruning the tree structure prior to searching the tree. This pruning reduces the cost of the algorithm to a polynomial function of the number of input microoperations. However, the reduction in cost also produces less than optimal microcode. [Ref. 8: p. 424]

List scheduling searches through each SLC segment and attempts to schedule each microoperation at the earliest possible point within the window of code that is being

considered. The size of the window is variable but the larger the window the longer the time required to do the job. Also, as the window size is increased, there is a diminished return (diminished amount of code compaction) for each unit increase in window size because of the increased chance of finding a data dependency. The further away the compaction is attempted, the greater the chance of two data items needing the same register, or some other data dependency. List scheduling is not optimal, but the cost is as low as first-come first-serve and the results are better than first-come first-serve.

Of these four local methods, only list scheduling and first-come first-serve can be done in what is considered a 'reasonable' amount of time and produce acceptable results. The fact that list scheduling produces better results than first-come first-serve in general was shown in a study done by Davidson, et al. [Ref. 8: p. 460] This would justify the use of list scheduling as the compaction method for the AMGS if only local compaction methods were available, however there are global compaction techniques that should be considered. It is an intuitive notion that global compaction techniques should provide better compaction since they look at the entire program and not only at small SLC segments.

It is true that, in general, global compaction techniques provide better compaction than local compaction

techniques yet, in comparison to local compaction techniques, global compaction techniques are very expensive. Trace scheduling, tree compaction, and compaction based on a generalized data dependency graph (GDDG) are the three most promising global compaction techniques. Trace scheduling identifies the most frequently traversed path through a section of microcode and does a local compaction on that path. The process is repeated on all of the paths through the microprogram until no further microoperation movement is possible. A data dependency graph must be constructed for each path analyzed and any microoperations that are moved must be documented. This documentation is done to insure that the moving of microoperations will have no effect on other loops. The bookkeeping for trace scheduling is the most expensive part. In fact in the worst case, the memory required to run a trace scheduling compactor can grow exponentially. [Ref. 9: p. 480] Therefore, although trace scheduling does an excellent job of microcode compaction, the overhead is too high to justify its use.

Tree compaction is based on trace scheduling. The advantage of tree compaction over trace scheduling is the control of the increase in memory size. Tree compaction divides the microprogram into subsets and applies the trace scheduling techniques to the subsets individually. This achieves compaction that is close to the results achieved

by trace scheduling yet is not nearly as expensive. This method may be useful when it is fully researched and understood, however tree compaction still produces microcode that is less than optimum and the cost can be high.

The third global compaction method is based on a global data dependency graph (GDDG). A GDDG "is capable of representing in a single chart the data dependency of microorders not only within a basic block but in different basic blocks." [Ref. 10: p. 924] Both trace scheduling and tree compaction use a data dependency graph (DDG) to represent the data dependency of microorders in the basic blocks, however a DDG is not capable of representing the data dependencies beyond the basic block. This is the most important aspect of global compaction; moving microorders to adjacent blocks when possible.

Through use of the GDDG, it is possible to identify microoperations that 'must' be in a basic block and those that 'may' be in a basic block. Then, by identifying the frequency of execution of the separate blocks it is possible to make intelligent choices about moving microoperations from block to block or within the same block. The algorithm costs an amount which "is practically O(n), where n is the number of microorders contained in a source microprogram." [Ref. 10: p. 930] This is a very low cost and the preliminary results show that the algorithm

provides compaction that is within three to five percent of optimum (handwritten) microcode.

Of the three global compaction methods described, only the method based on an GDDG is efficient and results in low costs. Why then did JRS not use this compaction method in the AMGS? The answer is that during development of the AMGS, this compaction method was not available. JRS is currently revising the system to incorporate the GDDG global compaction technique, which should result in a much more efficient system than was evaluated in this study.

By looking at the two main compaction methods, global and local, it is evident that global compaction holds the most promise for efficiency that approaches the optimum. Once global compaction methods are more thoroughly researched and developed, they will become the logical choice if the cost can be controlled. Global methods are the only methods that approximate the handcoded versions. Local compaction does provide some compaction but does not in general do as well as handwritten microcode.

D. AMGS LIMITATIONS

The AMGS developed by JRS is designed to allow a small CPU intensive algorithm to be compiled in microcode and placed in the WCS of the VAX 11/780. When the algorithm is needed it can be called from a Fortran program. There are several limitations of the system that are important to

remember when considering what applications may be used on this system. Individually the limitations may seem small and even unimportant, however, the combined effect of the limitations may eliminate some of the applications.

First, the WCS only has 1K words of memory for the microcode. Since the microcode must be loaded into the WCS before execution due to linkage requirements, paging of the algorithm into the WCS during execution is not considered an option. Therefore the user is limited to an algorithm or collection of algorithms that is no larger than 769 microinstructions because the other 255 instructions are used for predefined functions. In fact, of the 769 microwords of memory available, about 30 instructions are already taken up by function entry and exit code that is required for register initialization and can not be modified by the user. The exact number of instructions varies depending upon the microprogram being executed.

Compacting a long algorithm to fit into the limited space of the WCS may be difficult or even impossible. Once the user has determined that the algorithm will fit in the WCS, then he/she must determine the 'hot' spots of the program (portions of the algorithm that use the most CPU time), separate those parts of the program from the rest, code those parts in the JRS HLL, and set up the microcode procedure call. This may be only a minor inconvenience but, it is extra effort needed to use the AMGS.

Second, JRS claims that the AMGS code will do integer arithmetic and comparisons very quickly, but any problem involving primarily floating point arithmetic will achieve minimal, if any increase in performance. This is because the JRS HLL uses the same floating point acceleration routines as the Fortran program. Portions of the floating point algorithm that do not use the floating point accelerator may execute faster when executed on the AMGS, but the net gain will probably not be very large due to the overhead of the floating point accelerator. During the testing of the AMGS the truth of this claim by JRS will be documented since there will be several tests to check the floating point accelerator performance.

The JRS HLL is set up to support only integer and floating point data structures. No character data structure is available and therefore applications using characters are not considered feasible. Arrays of integers and floating point numbers are possible but the lack of a character data structure will limit some applications or at least make them very difficult to do.

If the algorithm includes I/O then the algorithm must be rewritten to eliminate the I/O from the portion of the algorithm to be coded in JRS HLL since the HLL does not include any I/O statements. The I/O can normally be moved into the Fortran program that will call the WCS program. Besides providing an I/O function, the Fortran program will

set up any data structures needed for the program. This is really no more than a minor inconvenience, but it does complicate the use of the system.

Several other restrictions are listed in the AMGS manual and repeated below.

1) Combined maximum of fourteen arrays and compiler temporary variables.

2) Maximum of twenty DO-loops nested at any one time.

3) Maximum of five hundred symbols may be defined in a program.

These restrictions will not, in general, eliminate applications but they are restrictions based on the implementation of the system on the VAX 11/780. These restrictions are important because they point out some of the machine dependencies that exist even when an attempt is made to remain machine independent.

The final limitation of the JRS AMGS is a simple observation. One of the main motivations for having an AMGS is to allow for portability of the microcode. Presently, this system is only implemented on the VAX 11/780. Therefore, a current, yet hopefully temporary limitation is that the AMGS has not been programmed to generate microcode for any other machines. This limitation will result in eliminating many of the advantages of the AMGS if it is not corrected.

Assuming the application algorithm can be coded around these limitations, the user should be able to achieve better throughput by using the AMGS. A goal of this project is to make it easier for a user to determine if a potential application will benefit from the use of the AMGS.

E. PERFORMANCE EVALUATION METHODOLOGY

The performance evaluation was conducted to determine the throughput possible using the AMGS. There are many techniques available for doing performance evaluations including hand timing, formula methods, instruction mixes, and benchmarks, each having individual advantages and disadvantages. The method used for this evaluation must be capable of comparing two different programs and of giving accurate results. Therefore a collection, or benchmark of programs was defined with each program representing a different possible application for the AMGS.

This kernel of programs was carefully developed to contain the characteristics of the many possible algorithms which might be run on the system. This is a very important step for the validation of the results. If the proper program characteristics are not tested, the results will be invalid. By categorizing the algorithms according to application it is possible to specify what applications will benefit by use of the AMGS.

After defining a kernel of programs and coding them in both Fortran and the JRS HLL, the programs were run and the results compared. Besides comparing execution time, other factors previously discussed in this chapter were considered. Ease of programming, system reliability, and the compatibility of the application problem with the AMGS were also considered.

One important question is how much better a manual microprogrammer could do. The purpose of using the AMGS is to achieve increased throughput without using a large amount of programming time as would be required with the manual method. Even though manual microprogramming is costly due to development time, it is considered the standard and the results of the performance evaluation should be compared against the standard. By comparing all three execution times, Fortran, JRS microcode, and hand written (actually hand compacted) microcode, it will be possible to identify the best applications and possibly determine methods for making the slower applications faster.

F. BACKGROUND SYNOPSIS

Since the main factors affecting the AMGS have been reviewed, the next step is to determine the kernel of programs to be tested. These programs must be representative of the applications that might be used on

the AMGS. The purpose of defining this kernel is to attain general results that will give an AMGS user an idea as to the effectiveness of a specific application. The next chapter will discuss the applications to be tested and the programs used to test those applications.

III. PROGRAM GENERATION

There are many limitations that must be considered when choosing the proper benchmark for a system. The benchmark must take into consideration the AMGS limitations enumerated in the previous section and insure that the results are not biased by those limitations. Limitations such as the WCS size and the existence of only integer and real data structures have a major effect on the applications possible when using the AMGS. With these limitations in mind, it is possible to define some applications that can use the AMGS. One common computer application that will definitely not have increased throughput due to AMGS use is I/O intensive applications. The HLL was designed without I/O capability because microcode implementations do not increase the throughput for I/O intensive applications. However there are several applications for which the AMGS should theoretically provide increased throughput.

The applications tested in this study are grouped into four basic areas. These areas are:

1) Integer mathematics
2) Floating point mathematics
3) Sorting and Searching (Comparisons)
4) Bit manipulations

There are several subcategories in the four basic areas. A discussion of the subcategories follows.

Mathematically intensive applications that do calculations within the limits of the AMGS are prime candidates for the system. There are several different types of mathematical calculations that should be considered. Integer arithmetic must be considered separately from floating point arithmetic due to the different methods used for doing the calculations. Integer addition/subtraction is handled internally by the AMGS, but the floating point accelerator (FPA) on the VAX computer is used for floating point calculations and integer multiplications. This call by the AMGS to the FPA results in a significant amount of overhead for each call. When a Fortran program calls the FPA there is also some overhead, but since Fortran translates to machine code and machine code calls to the FPA involve less overhead than AMGS calls, the net result is slower execution time for the AMGS code than for Fortran code during floating point operations. This extra overhead in the AMGS is due to a requirement to save the state of the microprogram prior to executing the floating point operation. The result is a net loss of throughput when doing floating point calculations or integer multiplication on the AMGS.

Several types of calculations are possible when doing integer and floating point calculations. Division,

multiplication, addition, and subtraction are different arithmetic operations and the increase in throughput may be different for each type of calculation. As much as possible, this project will categorize the different calculations and show the percentage of increase possible for each category, however, in the interest of reducing the total number of tests we will combine tests that are very similar. Since addition and subtraction take the same amount of time in microprogrammed processors, they will be tested together. Multiplication and division are not implemented similarly and will not be tested together. In fact, since division can usually be implemented as reciprocal multiplication, division will not be tested. Integer exponentiation is normally accomplished by a series of multiplications and therefore will be considered a part of the multiplication test.

Another major application of the AMGS is sorting and searching. Since sorting and searching both include comparisons of bit patterns, they may be considered together in one broad category. The major difference is that sorting usually includes moving of data or moving the pointers to the data, while searching simply involves comparing until the desired data is found.

One final category that is directly applicable to the NRL problem is bit manipulation. This category includes the comparing, shifting, and replacement of bits or fields

of bits within a word. This category may be an excellent application of the AMGS due to the bit manipulating commands that are built into the JRS HLL such as the shift, swap, and mask functions. Fortran has the ability to do the bit manipulations, but the functions are provided through library calls which tend to be slower than direct language implementation constructs.

The next section of this chapter is an explanation of each test and the basic area it is designed to test. The explanation of the results of each test is included in Chapter Five. Table 3-1 lists the four basic areas and the tests that cover each area.

Table 3-1: Specific Tests

Integer Math

1. Do Loop
2. While Loop
3. Summation
4. Factorial

Floating Point Math

1. Chebyshev Cosine
2. Fast Fourier Transform

Sorting/Searching

1. Bubble Sort
2. Sieve of Eratosthenes
3. Quicker Sort
4. Binary Search

Bit Manipulations

1. Bit Manipulation
2. Bit Reversal

The simplest test was designed using the loop structures. The WHILE loop and the DO loop provide a method for testing addition or multiplication and comparing the results directly with the Fortran equivalent. The

simplest test is a WHILE loop that only increments the loop counter. This test can be done as many times as the user desires and it also can be nested to any desired level to test the effect of nesting. The basic area being checked in this test is the addition and comparison required each time a loop is completed. This comparison is required to determine the test condition for exiting the loop. A DO loop is another version of the loop construct, with the increment being automatic and the condition test a part of the DO statement. By using these two tests it is possible to document how much time is required to execute the overhead steps in any loop. This overhead cost will be used to analyze programs with loops.

The next two tests use the basic loop structure to determine the summation of an integer or the factorial of an integer. Each of these tests can then be used with the results from the previous test to determine the amount of time required to do either an integer multiplication or an integer addition by simply subtracting out the loop overhead.

Floating point multiplication is the subject of the next test. By implementing a Chebyshev approximation for the cosine of an angle and calculating many values, it is possible to determine the amount of time spent doing floating point multiplication for each system. There are some floating point additions that will add overhead to the

test, but the effect of the additions should be minimal. This test particularly reveals the overhead of calling the floating point accelerator from the AMGS. JRS documentation states that since both Fortran and the AMGS use the same FPA, there should be no speed gained by use of the AMGS. If the overhead of calling the FPA from the microcode is too high, then it will make the AMGS slower than the Fortran. This is an important experiment since the results will be a prime factor in determining if the AMGS should be used for floating point applications.

There were three tests written to evaluate the ability of the system to do comparisons. The first is a sort algorithm called Quickersort written by R. S. Scowen. This algorithm works by continually splitting the array of values to be sorted into parts and sorting the parts using the same method. The second algorithm is a method to determine all of the prime numbers between two values. This problem, called the Sieve of Eratosthenes, uses additions, comparisons, and assignment statements to determine the prime numbers in a specified range. This algorithm will give an insight into how all three of these items interact to affect the throughput of the AMGS. The third test is a bubble sort that sorts an array of integers into ascending order. By using a loop construct, comparisons, and a simple assignment statement, this algorithm is an excellent example of a well structured

module that does comparisons and uses assignment statements.

Another test algorithm is a Fast Fourier Transform (FFT) written in two parts because the entire program would not fit into the WCS. One part is a bit reversal program that simply assigns elements of an array to different locations in the array. The other part is the complex multiplication plus a Chebyshev cosine and sine generation routine for use in the FFT. The bit reversal is an excellent comparison of assignment statements between the two methods and therefore goes in the bit manipulation category. The FFT complex multiplication is another floating point multiplication and addition algorithm. By using the results of these two algorithms, we gain an example of a long algorithm that uses the entire WCS (the FFT) plus an algorithm that is only concerned with moving values around in memory (the bit reversal).

The final test is an algorithm to do bit manipulations using the bit manipulating functions provided by both the JRS HLL and the Fortran library. The algorithm takes an array of integers and performs different operations on the integers such as AND, OR, EXCLUSIVE OR, etc. These operations were chosen directly from the example NRL source code, so this test specifically tests the NRL application.

With the test programs now fully defined, the next step is to describe the test runs and the timing mechanism used

to perform the tests. The interdependence of tests will be discussed in the next chapter as well as the effect of using different language features on the individual tests. After discussing these effects the test results can be presented and analyzed.

IV. TESTING

The testing of the programs was done with the most accurate tool available so that any error in the timing mechanism would be minimized. That is why the timing mechanism and its accuracy were so important to the results of this study. Once the accuracy of the timing mechanism was determined, the minimum length of the test was specified to make the test length much longer than the possible error. Besides the testing mechanism, there are other aspects of program design that affect the execution time of the resulting object code. Since this is primarily a comparison between Fortran and the AMGS, both the factors affecting the execution time of compiled Fortran code and the factors affecting the AMGS were identified and considered during the programming phase of the project. The desire was to make the tests as equitable as possible in the two different languages.

A. TIMING MECHANISM

The VMS system library provides a software mechanism for timing blocks of code. There are no hardware monitors available to time individual programs and hand timing is very inaccurate in a virtual memory system. The only method that is relatively accurate, easy to use, and can

account for the virtual memory mechanism is the system library timing function. There are two ways to use this library function and both methods display precision to the nearest one-hundredth of a second. The system manual claims that actually calling the system library timing function is more accurate than using the SECNDS Fortran language feature (which uses the system library function). [Ref. 11, p. C-30] Even though the claim of better accuracy is not substantiated by any specific numbers in the manual, the system library function was chosen for these tests.

There is a certain amount of overhead as a result of each library call and since this overhead cannot be accurately measured, it results in inaccuracy which must be minimized. To time a segment of code requires two calls to the library routine with the code to be timed sandwiched between the two calls. The first call starts the timing and the second call records the time. To minimize the impact of the overhead in each use of a library function, the minimum time for the code segment execution must be much longer than the overhead. For this study, we determined by actually testing a series of timing calls that the upper limit of the overhead for each library call was less than .005 seconds. Therefore, we designed the Fortran version (without common or subroutine) of each test to last a minimum of two seconds. This means that the

overhead for the two library calls in that version is less than one-half of one percent of the test length. All tests lasted longer than one second except for one test (the binary search microcode compacted version) and therefore the possible error due to the timer is less than one percent except in the one test that is shorter than one second.

Because some of the algorithms being tested can be accomplished very quickly (in less than 0.5 seconds) it is important to increase the execution time. This was done by repeating the algorithm several times to insure that enough time was spent in the algorithm to produce accurate results. To accomplish this, the input data can not be changed during the program iteration and all iterative counters must be reinitialized on each iteration. These extra instructions do add overhead to the test but the overhead is the same in each version of the test and therefore the impact was considered to be minimal.

When the timing mechanism is invoked it produces any of five different values that are useful in analyzing the amount of time spent in an algorithm. The first value available is the elapsed time spent in the system, whether executing or waiting. The second value is the total CPU time that the algorithm being timed was executing. This is the most important value since it displays the actual CPU time the program required to execute. Next is the number

of buffered I/O requests and the number of direct I/O requests. These numbers are not important in this study since no I/O is being done during the timing periods. The last available value is the total number of page faults occurring during the timed period. This number is valuable because it states how many times the job was interrupted and waited for a new page of memory to be fetched. The larger this value, the greater the chance for error because the clock must be stopped and started for each page fault. The fewer page faults and the closer the elapsed time is to the CPU time, then the less chance of inaccuracies due to timing errors.

B. LANGUAGE FEATURES AND THE EFFECT ON TIMING

Before looking at the effects of the language features it is important to note that if a programmer does 'dumb' things, almost any algorithm can be programmed inefficiently in any language. It is a basic assumption during these tests that the algorithms are not being programmed poorly and every effort is made to use good, solid algorithms. Also, since the same algorithm is being programmed in both languages, any bad programming practices will be present in both versions and therefore tend to cancel each other out.

The next consideration is the effect of language features on execution speed. In the JRS HLL, there are no

features that will affect execution time except for the call to the FPA when doing floating point arithmetic. Floating point arithmetic requires a separate algorithm because of the data representation required. The data must be represented in one word and that one word includes both a mantissa and an exponent. The algorithm must separate the mantissa and the exponent, perform the operation after aligning the decimal point, and then store the mantissa and exponent back into the single word of memory. Floating point arithmetic is common in all block structured arithmetic languages and therefore Fortran has the same problem, but not to the same extent as the JRS HLL.

The Fortran language is not as simple as the JRS HLL and therefore some of the Fortran language features affect the execution speed of the program. Fortran has several different data access and parameter passing modes that do affect the execution time of a program. It is important to design tests that show the effects of different uses of these features on the execution time of the same algorithm. Otherwise, the results of the tests will only be valid for the language features being used in that specific test and could not be generalized for any program in the testing category.

Some of the language features of Fortran that affect the execution time are common blocks, subroutines with common blocks, and subroutines with parameters. Common

blocks affect the execution time of a program because the blocks are placed in a specific location in memory which results in more indexing and slower data access for each item. If instead of using a common block the data is simply declared in memory, there will be a shorter access time for each data item and faster execution.

The use of subroutines adds overhead due to the linkage conventions and activation record initialization that is required. Each time a subroutine is called, the current state (registers and program counter) must be saved in an activation record to insure that the state can be reinitialized when the subroutine is exited. When common blocks are combined with the use of subroutines there is both the overhead of the subroutine call and the overhead of accessing the data items in the common data area. These two added types of overhead result in increased execution time when compared with code that does not use the features. On the other hand, the features provide methods of passing data that are not otherwise available. Therefore the user must tradeoff modularity in design and ease of passing data between subroutines for longer execution times.

The use of subroutines with parameter passing results in even more overhead because of the requirement to set up the data area for the parameters, passing the parameter upon subroutine call, and returning the new values of the

parameters upon subroutine termination. Again, this language feature adds to the convenience and modularity of the program, yet results in longer execution time.

It should be noted though that without common blocks or parameter passing there is no way to pass data between subroutines. Also, if a data area is large, parameter passing may be very costly, even to the point of being unusable. Another possibility is writing the program without using subroutines or data passing mechanisms. However this usually results in programs that are difficult and expensive to read and maintain. Since this is unacceptable for most software projects, most programs are written using some, if not all of these features.

In order to document this tradeoff, all programs were tested in each of the following categories.

1) Fortran without use of a common block or subroutine.
2) Fortran using a common block but no subroutine.
3) Fortran using a common block and a subroutine.
4) JRS HLL using a common block and a subroutine.

By testing each program using each of these methods, we can identify the amount of time added by the use of each language feature. The user can then weigh the use of the JRS HLL depending upon what features are desired. The most realistic comparison is between a Fortran program using subroutines with common blocks and the JRS HLL because the JRS HLL requires the use of both a subroutine call and a

common block. Besides, most large Fortran programs are written using subroutines and common blocks so that the resulting program is modularized yet allows for easy data access.

One other requirement was determined during the testing due to the VMS operating system being a virtual memory system. When preliminary tests were made it was determined that other users seemed to have an effect on the execution time of the tests. Therefore, the tests were made under two different conditions. One condition was with the system clear of any other users. The other condition was with other users on the system. This was done to be able to document the difference, if any difference existed, and clear up the question about the effect of other users on the timing of a program. Chapter Five has a further explanation of the timing mechanism accuracy analysis.

The main emphasis during the writing of the tests was on making the programs equivalent. All versions of each algorithm must do each step the same way so that the comparison is fair yet the tests must be programmed as a 'normal' programmer would do it in that language. If a program is written in a special way that is known to be optimal for one of the languages then the comparison of execution times would favor that language. However, if it is natural to use the feature in that language then that was the way it was done. One example of this policy is in

testing the cosine function. Since the VAX 11/780 VMS operating system provides a cosine library function, the library cosine function was compared to the Chebyshev approximation to see which method is faster. Thus both methods (Chebyshev and the system library function) will achieve approximately the same answer, however the algorithm used to achieve the answer will be different. This special case is done to measure the effect of not having a trigonometric function procedure available in the JRS HLL library. Included in this test is the resulting inaccuracy of the Chebyshev approximation, the space used to store the routine in the WCS, and the execution speed.

With the specification of the testing methods, testing categories, and timing mechanisms, the next step is to compare the results. A comparison of execution times of each program in each category of testing was accomplished. During the explanation of the comparison, an analysis of the data and a summary of the results is given. This analysis will allow us to specify which applications the AMGS improves and which applications the AMGS does not improve.

V. PERFORMANCE COMPARISON

The results of the tests can now be analyzed and compared since the factors affecting microprogramming and the processes involved in testing have been reviewed. To insure that the analysis is complete, the raw data is presented first followed by an analysis of the results. The analysis will first compare the effects of language features on each of the tests and then compare the results of the different types of tests (ie. while loop, do loop, etc.). The final section of this chapter discusses the validity of the tests and analyzes the error in the tests.

A. RAW DATA ANALYSIS

The raw data is given in Table 5-1. All tests were programmed in the four categories explained in Chapter Four but only five of the algorithms were hand compacted. The times shown in Table 5-1 are the mean values of ten tests of each algorithm without other users using the VAX 11/780. The number in parenthesis in the table, if shown, is the value that should be added or subtracted from the mean value to define the 99% certainty range for the mean value. If no number is shown in parenthesis, then the value is one hundredth of a second. An explanation of how the range was determined is given in the last section of this chapter.

Table 5-1: Test Results (in seconds)

PROGRAM	NO COMMON STRAIGHT FORTRAN	COMMON STRAIGHT FORTRAN	COMMON SUBROUTINE FORTRAN	JRS HLL	HLL HAND OPTIMIZED MICROCODE
While Loop	11.11	18.18	20.20	11.12	
Do Loop	7.07	10.10	12.12	10.12	
Factorial	4.61	7.46(.02)	7.54	8.88	8.63
Summation	4.49	9.77(.02)	9.77(.01)	5.70	2.87
Cosine	5.09	6.17	6.24	8.62	
Cosine(Lib)	8.72	9.43(.14)	----	----	
FFT	11.16(.02)	13.08(.02)	13.74(.03)	17.37(.02)	
Sieve	3.39	4.18	4.10(.02)	2.49	
Binary Search	2.50	3.54	3.43	1.17	0.79
Bubble Sort	3.59(.03)	4.58(.03)	4.79(.03)	3.77(.02)	2.29
Quicker Sort	8.78(.04)	9.75	9.80(.02)	4.75	4.36
Bit Reversal	4.75(.02)	7.49	7.50	2.25	
Bit Manip	8.40(.04)	8.53(.07)	8.41(.02)	2.98	

Not all programs were hand compacted because the compaction required special knowledge of VAX microprogramming and also required a significant amount of time. The tests to be compacted were chosen to insure that a representative sample was taken from each of the categories. Another criterion for choosing the tests for compaction was to choose some tests which were faster in Fortran and some tests that were faster in microcode to compare the effect of compaction. The basic purpose was to

see, in general, how much better we could do with the compaction without exerting a tremendous amount of effort. That purpose was attained by compacting the five selected programs.

When looking at the times from Table 5-1 in general, some of the results were counter-intuitive because the expected result is to have the microcoded version execute faster. In many cases the Fortran versions were faster or as fast as the microcoded versions. This can be attributed to three facts mentioned in earlier chapters. First, the microcode that is generated from the HLL by this AMGS is not compacted. Second, the Fortran compiler generates highly optimized code. The third reason is that some of the routines used as tests involve floating point arithmetic or integer multiplication, both of which use the floating point accelerator. The use of the floating point accelerator results in increased overhead for microcode. These three factors, separately or combined, resulted in some cases where the Fortran outperformed the microcode.

B. EFFECTS OF LANGUAGE FEATURES ON THE TESTS

The different Fortran language features were tested to isolate the effects of the different techniques for data passing. The important point is that the tests were programmed as a 'normal' programmer would program them. No special attempts were made to make specific tests run well

in either of the two languages. Since it was unlikely that a determination could be made as to what the "normal" programmer would do, the three different Fortran tests were devised so that the user could determine which method was needed for his/her application. Of course, if a programmer chose the Fortran without subroutines or common data areas, then he/she was giving up the use of some very important software engineering techniques.

In general, the tests of the different Fortran language features resulted in more speed with fewer features and less speed with more features. The fastest Fortran technique in all cases was the version that used no subroutines and no common data structure. The use of common data areas with and without subroutines resulted in somewhat unexpected data. The expected results were for the versions using common data without subroutines to run faster than the version using common data areas with subroutines. This occurred in most but not all of the tests. In general, there was only a slight increase in execution time when a subroutine with common was compared with the same program with no subroutines but with common data, which implies that the overhead of calling and returning from a subroutine (without any parameters) is not very significant. In fact, in most cases there was no statistical difference (discussed in the last section of this chapter) between the tests with subroutines and common

and the tests without subroutines but with common. One possible explanation is that the variation in the length of time to start and stop the timing mechanism is greater than the length of time required to call and return from a subroutine. Since there is only a single call in each test, the results may not show any difference when the subroutine is used.

The hand compaction of the JRS HLL microcode always resulted in faster execution than the uncompacted JRS HLL microcode. This is as expected since the hand compacted code was derived from the JRS HLL microcode. In no case were instructions expanded ("n" microinstructions encoded into "n+k" microinstructions, where k > 0) and therefore no increase in execution speed for the hand compacted code was expected. It should be realized that the method used for generating the hand compacted microcode does not really produce hand compacted microcode because the compaction was done to an existing program. The microprogrammer did not set up the problem according to his own liking. The microprogrammer simply took the generated microcode and compacted it using his knowledge of the VAX microprogramming. If microoperations could be combined with other microoperations to reduce the total number of microinstructions, they were combined. The important point to remember is that the microcode was machine generated and hand compacted.

Another point that must be mentioned about the data analysis in general is the overhead involved in the JRS HLL microcode. The length of time required to make the call to the microcode plus the overhead involved in the use of common data is not documented anywhere and can not be determined in this study because the timing mechanism is not accurate enough. Therefore during the analysis of the data, it is important to remember that when the JRS HLL microcode is called there is a certain amount of overhead in the call. This overhead is most likely more than the overhead of a subroutine call in Fortran because the state of the micromachine must be initialized. The other point is that all data in the microcode is in a common data area and therefore, as has been documented, requires extra time to access. Probably the best comparison between Fortran and JRS HLL is to use Fortran with common data and subroutines because the overhead of the common data and the subroutine calls approximately cancel out each other. Therefore, it is possible to compare the actual speed of each method rather than comparing the overhead involved in each method.

The overhead involved in the subroutine call and the common data area will not always be constant. If there are only a few data items being accessed in the subroutine then all of the data values can be placed in registers which reduces the access time. However, if an array or a large

number of variables are being accessed then it will take longer to get the data in and out because of the use of a common data area. The important point when looking at the comparisons being made in the next few sections is that if common data structures and subroutines are used in Fortran (which is almost always done) then the execution speed will not be as fast as the fastest Fortran test. If the decision is made to not use the common data structures and subroutines then the programmer will be giving up modularity of design and other software engineering techniques for faster execution.

C. COMPARISON OF TEST RESULTS

This section will compare the results of the Fortran versions with the HLL versions. The comparison will be done within the four basic areas defined in Chapter Three. Each test algorithm is available in an appendix in both the Fortran implementation and the JRS HLL implementation. The Fortran version of the algorithms available in the appendices is the version in a subroutine with a common data structure. The algorithms have been placed in the appendices according to the basic area that they are testing. Table 5-2 lists which appendices contain which individual tests. The algorithms have been removed from the individual test harnesses, however an example harness (Factorial Program) is available in Appendix E.

Table 5-2: Table of Tests in Appendices

Appendix A: Integer Mathematics

1. Do Loop
2. While Loop
3. Summation
4. Factorial

Appendix B: Floating Point Mathematics

1. Fast Fourier Transform
2. Chebyshev Cosine

Appendix C: Sorting/Searching

1. Binary Search
2. Quicker Sort
3. Sieve of Eratosthenes
4. Bubble Sort

Appendix D: Bit Manipulations

1. Bit Manipulation
2. Bit Reversal

1. Integer Mathematics

The basic loops were included in the integer mathematics category because all that occurs in the loop construct is an increment and test until the condition is met, at which time a jump out of the loop is executed. This is very simple and uncomplicated so the expectation was that the microcoded version would not be much better than the Fortran version. In fact, the JRS HLL WHILE loop was only as fast as the fastest Fortran version while the fastest Fortran DO loop was much better than the JRS HLL DO loop. The results imply that the Fortran code is highly optimized.

Since each of the loop tests involves only one variable, the common data area access time penalty can not be blamed since the variable was stored in a register. There is the overhead of calling the subroutine and setting up the data registers however that alone should not cause the microcode to be as slow or slower than Fortran. The only logical answer is that the optimization and compaction of the different codes has a large effect on the execution speed. One other important point about the loops is that in all cases the DO loop is faster than the WHILE loop. This is most likely due to better optimization because the looping variable is part of the loop construct while in the WHILE construct the incrementing of the variable occurs independently from the language construct.

The next test was the summation of an integer value. This test measured how fast addition could be done, however, since each summation could be done very quickly, a loop construct was set up to repeat the summation 10,000 times. The results were that the fastest Fortran version was slightly faster than the JRS HLL version, even after subtracting the overhead of the WHILE loop. This result was not expected but can be explained as the result of lack of code compaction because when the summation microcode was compacted by hand, the execution speed became significantly faster than the fastest Fortran version.

The final integer mathematics test is the factorial program. The test was limited to a maximum input of 12 because 13! is beyond the limits of the storage capacity of a four byte integer. Therefore, to make the test last long enough for timing purposes, a loop was set up to calculate the factorial 100,000 times prior to stopping.

The result of the factorial test validates the JRS claim that integer multiplication is slow because of the use of the FPA. After subtracting the overhead of the loop, the JRS HLL is still twice as slow as the fastest Fortran version. In fact, the slowest Fortran version, using common data areas and a subroutine, is faster than the JRS HLL microcode. Therefore, the AMGS should not be used for integer multiplication intensive algorithms because of the FPA overhead.

The JRS HLL did not result in any performance improvements for any of the integer arithmetic tests accomplished in this study. This was due either to a lack of microcode compaction or to the use of the FPA for integer multiplication.

2. Floating Point Mathematics

There were two algorithms for testing the floating point mathematics applications, the Chebyshev Cosine routine and the Fast Fourier Transform (FFT). Both algorithms substantiated the JRS claim that floating point calculations would not do well in microcode. The execution

speed of the FFT HLL version was about twice as long as the fastest Fortran version. The other Fortran versions were of course slower than the fast version due to the use of common and a subroutine call.

The Chebyshev Cosine routine gave the same type of results as were attained for the FFT, a slow down of about 80%, caused by the FPA. However, the interesting part of this test is in comparing the speeds of the Chebyshev Cosine with the speed of the Cosine Library function. The overhead of the Library Function call is very high because even the JRS HLL (which is the slowest Chebyshev version) is faster than the Library Function test. Therefore it is justifiable to say that while the use of the HLL for doing trigonometric computations is not a great improvement, this test does demonstrate that the commonly used features of a language can be costly and that the microcode does give slightly better performance than the Library Function.

Both tests in this basic area support the JRS claim that floating point arithmetic will not be helped when coded in JRS microcode. Since that point has been well documented, we will now look at the sorting and searching tests to see what kind of results they produce.

3. Sorting and Searching

There were four tests accomplished in this area and three of the four gave results that were favorable for the JRS HLL. The one test where the JRS HLL ended up being

slightly slower was the bubble sort algorithm. There was no looping mechanism to subtract away from the problem and the algorithm consists of only assignment statements and comparisons. Therefore there is no reason to explain the slow performance except for the lack of compaction of the microcode.

The Sieve of Eratosthenes program test resulted in the JRS HLL version running about 25% quicker than the fastest Fortran version. This result was expected since the microcode is able to do comparisons rather quickly. One other interesting point became apparent during this test. Since the tests are supposed to be written as a 'normal' programmer would write them, it is sometimes easier to use a DO loop rather than a WHILE loop or vice versa. However, when trying to get code to execute quickly, it is obvious that the Fortran DO loop is much faster than the Fortran WHILE loop as shown in Table 5-1. On the other hand, the JRS HLL DO loop is not nearly as fast as the Fortran DO loop and only slightly faster than the JRS HLL WHILE loop. Therefore, a program is dependent upon the language construct chosen by the individual programmer and if a DO loop is used in the Fortran version while a WHILE loop is used in the JRS HLL version, there will be a greater difference in results.

To avoid this discrepancy in results (after it was noticed in the initial results), the Sieve algorithm was

rewritten in both languages using DO loops because a definite iteration (the DO loop function) was what was needed in the algorithm. The change in speed of the algorithms due to the use of the DO loop was not tremendous. However, this test does demonstrate the effect of using different language constructs plus the use of 'normal' programming techniques and constructs.

The Quicker Sort algorithm demonstrated the speed of the microcode as was expected. Since only comparisons, additions, and subtractions with one multiply are used, this algorithm is very fast. The Binary Search algorithm results ended up with the JRS HLL being twice as fast as the fastest Fortran version. Again, this was expected because of the use of comparisons during most of the algorithm. This algorithm produced the second best performance increase by the JRS HLL microcode of all of the tests. This was probably due to the fact that the algorithm has only one DO loop, one WHILE loop, and the rest of the algorithm is made up of if-then constructs which are simple comparisons.

The sorting and searching tests were a good application of the JRS HLL microcode. For the most part, the microcode resulted in faster execution speed than the corresponding Fortran program, however, the increase was never much more than twice as fast.

4. Bit Manipulations

The last basic area of the tests is the bit manipulation area. Two tests were accomplished in this area and both gave positive results for the microcode version. The bit reversal program ended up with a large increase in execution speed. The program was simply used to switch items in an array. No comparing was needed since the program switched the items in the array according to a convolution scheme. This test demonstrates the speed of the assignment statement in the microcode.

The bit manipulation program also resulted in faster execution for the JRS HLL than for the fastest Fortran version. The main reason for this fast execution is that the Fortran version uses system library routines which are slow to call and execute. Therefore, it is actually the slowness of the Fortran library routine rather than the speed of the microcode that gives the increased throughput. The important point is that the microcode does improve upon the execution speed of the corresponding Fortran code and therefore the AMGS gives a performance increase for these kinds of operations. It is also important to note that this program was simply a series of calls to the microcode or Fortran routines that perform the functions. No other operations besides the driving DO loop were needed in the algorithm and therefore it was a very accurate test of the actual speed of the tested code.

D. TEST ERROR ANALYSIS

Because this testing was done on a virtual memory system, there was a possibility of error due to the timing mechanism being switched on and off many times. The intention of the tests were to give the user an accurate estimate of how much speed would be gained by using the AMGS. To insure that the estimate is as accurate as possible, a computation was made to determine the confidence interval for the mean. Also, to determine if the virtual memory system was affecting the results, a test was performed that allows us to state, with a specified amount of confidence, whether the virtual memory system affects the results.

Since we made several runs of each test, we were able to determine a mean execution time for each test and a standard deviation for each test. However it is important to do a statistical analysis to determine how confident we are of these results. The question of confidence was answered by using the Student T distribution (because of the small sample size) to find the interval within which the mean will fall with the specified amount of confidence. For these tests, a confidence of 99% was desired. The following formula was used to determine the range of the mean execution time for 99% confidence. The value for 't' is dependent on the level of confidence desired and was

read from a Student T distribution chart. [Ref. 12: p. 488] 'X' is the mean of the sample, 'S' is the sample standard deviation, and 'n' is the total number in the population.

$$X - t(S / n) < u < X + t(S / n)$$

To find the effect of the virtual memory system required performing each test under two different conditions. First, each test was made with other users on the system. This could be anywhere from one other user to twenty users. Next, each test was performed with all other users locked out of the system and the entire computer running only the system support programs and the tests for this project. Then a hypothesis, called the null hypothesis (H1) was assumed. The null hypothesis was that both samples came from the same population. To test the null hypothesis we used the following formula, where X1 and X2 are means, S1 and S2 are standard deviations, and n1 and n2 are sample sizes (in this test, 10).

$$t = (X1 - X2) / SQRT((S1/n1 + S2/n2))$$

If the calculated 't' (from above) >= 't' (from the chart based on 99% certainty), then the null hypothesis can be rejected. In other words, the samples do not come from the same population which means that the number of users on the system does affect the results. If the value 't' < 't' (from the chart) then they could be from the same population and the other users on the system may not affect the results. [Ref. 12: pp. 214 - 221]

The results of the first confidence test mentioned above were enumerated in Table 5-1 with data taken on the VAX 11/780 with all other users locked out of the system. In general, the results were very accurate in that they gave a small range in which the anticipated results would fall. The null hypothesis test gave mixed results. It was hoped that we would be able to state that the tests with other users on the system would be from a different sample set than the tests without other users. However, that was not the case in general. In most situations, the tests with other users on the system simply showed a higher mean but the possible range for 99% certainty included most or all of the range for the test without other users. Therefore, in the second test the null hypothesis could not be refuted in most cases. However, it does appear that other users on the system do affect the timing mechanism but only because they increase the standard deviation of the tests and thereby widen the range of values for 99% certainty.

VI. CONCLUSIONS

The purpose of this project was to evaluate the performance of the JRS AMGS. This has been accomplished by comparing the performance of the JRS HLL microcode with Fortran code on the VAX 11/780. The testing has produced some unexpected results and has shed light on several interesting points. The first point being that microcode will not always result in faster execution of an algorithm. During the testing it became apparent that this was due mainly to two causes. One reason is that for the speed of microcode to be fully utilized, the microcode must be properly compacted. The other is that the use of the FPA by the microcode results in slightly degraded performance.

The second point is the effect of the different language features upon the execution speed of the Fortran code. When the fastest Fortran code was compared with the microcode there were several cases where the Fortran was much faster than the microcode. However, when the slowest Fortran code was compared with the HLL microcode the microcode was faster. This was true in all cases except when the FPA was required. Testing the effects of the language features revealed an important point since the use of the features allows a programmer to use software engineering techniques. When these features are not used

it is very difficult for a programmer to use software engineering techniques such as modularity and information hiding. Without these techniques the code may run fast but it is usually very difficult to develop and always hard to maintain. Therefore, a tradeoff must be made between the convenience and security of the language features or the speed advantage possible without the features.

A. CONCLUSIONS FROM THE DATA ANALYSIS

The analysis of the data allows for some conclusions to be drawn about the use of the AMGS for specific applications. The conclusions are grouped in terms of the four general areas defined in Chapter Four rather than about individual tests so that a user may make a decision based upon a general category of application rather than a specific example program. Specific program results will be mentioned if the results of that test vary significantly from the other tests in the specific area being discussed.

The integer mathematics application resulted in no advantage from the use of the AMGS. This is most likely due to the lack of compaction of the microcode. This conclusion is justified because when the summation program's microcode was compacted and subsequently executed, the results were a significant increase in execution speed. Therefore, it is assumed that if the code was properly compacted the execution speed would be

improved. The only test in the integer mathematics category that would not be greatly improved by the compaction is the factorial test. This is due to the use of the FPA for integer multiplication.

The floating point mathematics area also turned out not to be a good application for the AMGS. This was expected and the probability that this would happen is documented in the JRS HLL manual. The difference in the magnitude of the execution speeds is interesting because the JRS HLL runs about 60% slower than the fastest Fortran version.

The sorting and searching application area demonstrated promising results for the AMGS. In three of the four tests the AMGS version was significantly faster than the fastest Fortran version. In one test (the bubble sort), the Fortran was faster than the AMGS but this is probably due to a lack of compaction rather than due to a lack of applicability to the AMGS. From the results of these four tests, it is justifiable to say that sorting and searching are both good application areas for using the AMGS. However, it should be noted that at this point in the AMGS development, the difference in execution speeds is not as good as it could be with compacted microcode.

The bit manipulation area also resulted in favorable results for the AMGS. In fact, this was the best applications area of the JRS HLL because both tests ended up more than doubling the speed of the Fortran code. Of

course, one of the tests was slow in the Fortran version because of the use of library functions, however, since that was the only way to easily perform that function in Fortran, that was the way it was programmed.

Now that we have defined the areas where improvement is possible the question remains about whether the AMGS should be used by NRL? The answer to this must be based on more factors than simply execution speed. We must also consider system cost, ease of use, and actual improvement possible.

Since the improvement is at the best two to three times better than the Fortran code, the cost in money and programming effort can not be justified by the possible gain. When the system is improved to include microcode compaction with a resultant increase in performance, then the AMGS cost may be justified. Somewhere in the area of an order of magnitude increase in speed is necessary before the cost of the system (money and programming effort) is justified.

The AMGS did prove capable of producing microcode that is as fast or slightly faster than the compiled Fortran code. Therefore, if an application exists that will use a microcoded machine, the AMGS is capable of producing a large amount of 'acceptable' microcode. The AMGS can produce the microcode very quickly in comparison to conventional methods. Also, the AMGS can produce large amounts of microcode at much less expense than is possible

with hand microcoding. The AMGS therefore provides a mechanism for producing 'acceptable' microcode efficiently and inexpensively.

One other possible use of the AMGS is to produce microcode that can be hand compacted. If the microprogrammers are available, the HLL can be used to produce an uncompacted microprogram and then the microprogrammers can be used to compact the HLL microcode. This technique produced very good results during the study and the cost in microprogrammer's time is much less than writing a complete microprogram from scratch.

B. FUTURE RESEARCH POSSIBILITIES

There are several areas that can be researched as a continuation of this work. Some areas relate directly with this type of microcode generating system but other areas are points that became obvious during the study yet had to be ignored to keep the scope of the thesis within reason. One area of research is to evaluate the next version of the JRS AMGS. The next version is now available and has microcode compaction which should result in much better run time results. Also, the revision has more language constructs that more closely parallel the constructs available in the more modern block structured languages. With these revisions, it should make the system easier to use and give better results.

Since one of the suggested advantages of the AMGS is portability of the JRS HLL microcode, it is important for this system to be implemented on another machine so that the work involved in doing such a job can be documented. The possibility of implementing the AMGS on another machine is already a stated goal, but until it is done, a proper testing of both implementations can not be made. The comparison of the results of the tests would document the portability of the system and demonstrate the ease with which the machine transition could be made. It would also be advantageous to have another language such as Fortran or Pascal used as the source code instead of the JRS HLL. This would make the AMGS accessible to more people resulting in a better chance of the system becoming more widely used.

The cost of using different language features in Fortran was interesting even though it was a sidelight of the study. Further study could be done as to the exact cost of using a subroutine with or without parameters. Also, the actual cost of using a common data area could be documented so that a user knows how much the use of such a feature is costing. Of course this kind of testing would be system dependent, but if that system used these language constructs for a significant amount of work, the results could be very helpful in making decisions during future programming efforts.

The final suggestion for further research has to do with defining the application areas. It would be very helpful if there were some guidelines as to what applications use what operations. These guidelines would be very helpful during future system performance evaluation efforts.

LIST OF REFERENCES

1. Sheraga, R. J. and Geiser, J. L., "Experiments in Automatic Microcode Generation", IEEE Transactions on Computers, June 1983, pp. 557 - 569.

2. Wilkes, M. V., "The Best Way to Design an Automatic Calculating Machine", Advances in Microprogramming, Artech House, 1983, pp. 58 - 60.

3. Rauscher, T. G. and Adams P. M., "Microprogramming: A Tutorial and Survey of Recent Developments", IEEE Transactions on Computers, January 1980, pp. 2 - 19.

4. Booch, G., Software Engineering with ADA, Benjamin/Cummings, 1983.

5. Davidson, S., "High Level Microprogramming - Current Usage, Future Prospects", Micro 16, October 1983, pp. 193 - 200.

6. Geiser, J. L., "On Horizontally Microprogrammed Microarchitecture Description Techniques", IEEE Transactions on Software Engineering, September 1982, pp. 513 - 525.

7. Gries, D., Compiler Construction for Digital Computers, Wiley, 1972.

8. Davidson, S., Landskov, D., Shriver, B. D., and Mallett P. W., "Some Experiments in Local Microcode Compaction for Horizontal Machines", IEEE Transactions on Computers, July 1981, pp. 460 - 477.

9. Fisher, J. A., "Trace Scheduling: A Technique for Global Microcode Compaction", IEEE Transactions on Computers, July 1981, pp. 478 - 490.

10. Isoda, S., Kobayashi, Y., and Ishida, T., "Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph", IEEE Transactions on Computers, October 1983, pp. 922 - 933.

11. VAX-11 Fortran Language Reference Manual, Digital Equipment Corporation, No. AA-D034C-TE, April 1982.

12. Miller, I. and Freund, J. E., Probability and Statistics for Engineers, Prentice-Hall, 1972.

APPENDIX A

INTEGER MATHEMATICS ALGORITHMS

```
C THE DO LOOP IN A FORTRAN SUBROUTINE
C 'I' IS THE LOOP VARIABLE WHILE 'K'
C IS THE TOTAL NUMBER OF TIMES THE LOOP
C WILL BE EXECUTED.

      SUBROUTINE DOLOOP

      COMMON /NOS/ I,K

      DO I=1,K

      ENDDO

      END     ! OF DOLOOP
```

```
\ THIS PROGRAM IS A DO LOOP WRITTEN IN JRS HLL \
    PROGRAM DOLOOP;
    INTEGER I,K;
            DO I = 1 TO K;
            ENDDO;
    STOP;
    END.  \ OF DOLOOP \
```

```
C THIS IS THE WHILE LOOP IN FORTRAN
C COUNT HOLDS THE TOTAL NUMBER OF TIMES
C THE LOOP WILL BE EXECUTED.  ZERO HOLDS
C THE VALUE 0.

      SUBROUTINE WILELOOP

      INTEGER COUNT, ZERO

      COMMON /WCS/ COUNT, ZERO

            DO WHILE (COUNT .GT. ZERO)

                  COUNT = COUNT - 1

            END DO

      END  ! OF WILELOOP
```

```
\ THIS IS THE WHILE LOOP IN JRS HLL \
     PROGRAM WHILELOOP;
     INTEGER COUNT, ZERO;
             DO WHILE (COUNT .GT. ZERO);
                COUNT = COUNT - 1;
             ENDDO;
     STOP;
     END.
```

```
C THIS IS THE SUM ALGORITHM IN FORTRAN
C 'COUNT' IS THE NUMBER OF TIMES THE SUMMATION WILL
C BE COMPUTED.  'VALUE' IS THE NUMBER TO BE SUMMED.
C 'TEMP' IS A STORAGE LOCATION FOR 'VALUE'.  'TOTAL'
C IS THE VALUE OF THE SUMMATION.  'ZERO' HOLDS THE
C VALUE 0.

      SUBROUTINE SUM

      INTEGER TOTAL, VALUE, TEMP, COUNT, ZERO
      COMMON /NCS/ TOTAL, VALUE, TEMP, COUNT, ZERO

      ZERO = 0

      DO WHILE (COUNT .GT. ZERO)

C REINITIALIZE THE VARIABLES FOR THE SUM ROUTINE

         COUNT = COUNT - 1
         VALUE = TEMP
         TOTAL = ZERO

C THIS IS THE ACTUAL SUMMING OF THE VALUE

         DO WHILE (VALUE .GT. ZERO)
            TOTAL = TOTAL + VALUE
            VALUE = VALUE - 1
         END DO

      END DO

      END ! OF SUM
```

```
\ SUMMATION ALGORITHM IN JRS HLL \

     PROGRAM SUMMATION;

     INTEGER TOTAL, VALUE, TEMP, COUNT, ZERO;

     DO WHILE (COUNT .GT. ZERO);

        COUNT = COUNT - 1;
        VALUE = TEMP;
        TOTAL = 0;

        DO WHILE (VALUE .GT. ZERO);
           TOTAL = TOTAL + VALUE;
           VALUE = VALUE - 1;
        ENDDO;

     ENDDO;

     STOP;
     END.
```

```
C THE FACTORIAL SUBROUTINE IN FORTAN
C 'COUNT' DETERMINES HOW MANY TIMES THE FACTORIAL
C OF 'VALUE' WILL BE DETERMINED.  'TOTAL' HOLDS
C THE ANSWER AND IS INITIALIZED TO 1.  'TEMP'
C HOLDS THE FACTORIAL VALUE TO BE DETERMINED
C FOR REUSE.

      SUBROUTINE FAC

      INTEGER TOTAL, VALUE, TEMP, COUNT, ZERO, ONE
      COMMON /NOS/ TOTAL, VALUE, TEMP, COUNT, ZERO, ONE

      DO WHILE (COUNT .GT. ZERO)

         COUNT = COUNT - 1
         VALUE = TEMP
         TOTAL = ONE

         DO WHILE (VALUE .GT. ZERO)
            TOTAL = TOTAL * VALUE
            VALUE = VALUE - 1
         END DO

      END DO

      END ! OF FAC
```

```
\ THE FACTORIAL PROGRAM IN JRS HLL \

    PROGRAM FACTORIAL;

    INTEGER TOTAL, VALUE, TEMP, COUNT, ZERO, I;

    DO WHILE (COUNT .GT. ZERO);

       COUNT = COUNT - 1;
       VALUE = TEMP;
       TOTAL = 1;

       DO WHILE (VALUE .GT. ZERO);
          TOTAL = TOTAL * VALUE;
          VALUE = VALUE - 1;
       ENDDO;

    ENDDO;

    STOP;
    END.
```

APPENDIX B

FLOATING POINT MATHEMATICS ALGORITHMS

```
      SUBROUTINE FFT
C
C     ******************************************************
C     FAST FOURIER TRANSFORM
C     ******************************************************
C
C     X       -  COMPLEX ARRAY X(2**M)
C
C     M       -  ORDER OF FFT, N=2**M
C
C
C        BASED UPON AN FFT FIRST DEVELOPED BY SIGNALS
C     SCIENCE CORPORATION FOR PROJECT SALESCLERK.
C
C        FIRST TRANSCRIBED BY LCDR C LAURVICK, USN
C        MODIFIED BY LT M HARTONG, USN
C
      REAL XREAL(4096),XIMAG(4096),TREAL,TIMAG,T2REAL,
     1        T2IMAG,WREAL,WIMAG
      DATA PI/3.14159265/
      N=2**M
C
C     M STAGE FOURIER TRANSFORM
C
      DO 20 L=1,M
      LE0=2**(M+1-L)
      LE1=LE0/2
      UREAL = 1.0
      UIMAG = 0.0
      PHASE = PI/FLOAT(LE1)
C
C     W=CMPLX(COS(PHASE),-SIN(PHASE))
C
          IF (PHASE .GT. (PI/2.0)) THEN
               PHASE2 = PI - PHASE
          ELSE
               PHASE2 = PHASE
          ENDIF
C
          COSX = 0.99995795 -(0.49924045 *PHASE2 * PHASE2)
          COSX = COSX + (0.03952674 * PHASE2 * PHASE2 *
     1         PHASE2 * PHASE2)
C
          IF ( PHASE .GT. (PI/2.0) ) THEN COSX = -COSX
C
C CALCULATE SIN
```

```
      IF (PHASE .LT. (PI/2.0) ) THEN
         PHASE2 = PI/2.0 - PHASE
        ELSE
            PHASE2 = PI - (3.0 * PI/2.0 -PHASE)
       ENDIF

      SINX = 0.99995795 -(0.49924045 *PHASE2 * PHASE2)
      SINX = SINX +(0.03962674 * PHASE2 * PHASE2 *
     1        PHASE2 * PHASE2)

C     DECIMATION IN TIME

      DO 20 J=1,LE1
      DO 10 I=J,N,LE0
      IP=I+LE1

         TREAL = XREAL(I) + XREAL(IP)
         TIMAG = XIMAG(I) + XIMAG(IP)

         T2REAL = XREAL(I) - XREAL(IP)
         T2IMAG = XIMAG(I) - XIMAG(IP)

         XREAL(IP) = T2REAL * UREAL - T2IMAG * UIMAG
         XIMAG(IP) = T2REAL * UIMAG + T2IMAG * UREAL

         XREAL(I) = TREAL
         XIMAG(I) = TIMAG

10       CONTINUE

         UREAL = UREAL * COSX - (UIMAG * -SINX)
         UIMAG = (UREAL * -SINX) + UIMAG * COSX

20    CONTINUE
99    RETURN
      END
```

```
      PROGRAM FFT;

\     ************************************************\
\     FAST FOURIER TRANSFORM  - AMGS HLL VERSION     \
\     ************************************************\

      INTEGER I,J,M,N,L,LE0,LE1,IP,KOUNT,K,NV2,NM,NM1,P;

      REAL UREAL,UIMAG,PHASE,COSX,SINX,TREAL,TIMAG,
      1         T2REAL,T2IMAG;
      REAL TMP,PI,R1,R2,R3,K3;

      REAL ARRAY XREAL(4096),XIMAG(4096);

\ DOES NOT DO BIT REVERSAL                                  \

\ N = 2**M                                                  \

      N = 1;
      DO KOUNT = 1 TO M;
            N = 2 * N;
      ENDDO;

\       M STAGE FOURIER TRANSFORM                           \

\       EXECUTE THE LOOP 10 TIMES FOR TIMING PURPOSES       \

      DO K = 1 TO 10;

      DO L=1 TO M;

\ REPLACE WITH INLINE EXPANSION DUE TO NO EXPONENTS         \
\       LE0=2**(M+1-L);                                     \

         LE0 = 1;
         DO KOUNT = 1 TO (M+1-L);
           LE0 = 2*LE0;
         ENDDO;

       LE1=LE0/2;
       UIMAG = 0.0;
       UREAL = 1.0;

       PHASE=  PI/FLOAT(LE1);

\       W=CMPLX(COS(PHASE),-SIN(PHASE))\
\          GENERATE SIN AND COS                \


          IF (PHASE .GT. (PI/2.0)) THEN
            DO;
```

```
            TMP = PI - PHASE;
          ENDDO
          ELSE
           DO;
           TMP = PHASE;
          ENDDO;


        COSX = R1 -  (R2 * TMP * TMP ) +
     1        (R3 * TMP * TMP * TMP * TMP);

        IF (PHASE .GT. (PI/2.0)) THEN
          DO;
          COSX = -COSX;
          ENDDO;

    \ CALCULATE SIN \

       IF ( PHASE .LT. (PI/2.0) ) THEN
          DO;
           TMP = PI/2.0 - PHASE;
          ENDDO
          ELSE
           DO;
           TMP = PI - ( R3 * PI/2.0 - PHASE);
       ENDDO;

       SINX = R1 -  (R2 * TMP * TMP ) +
     1        (R3 * TMP * TMP * TMP * TMP) ;


\      DECIMATION IN TIME                                \

       DO  J=1 TO LE1;
         DO  I=J TO N BY LE0;
               IP=I+LE1;

               TREAL = XREAL(I) + XREAL(IP);
               TIMAG = XIMAG(I) + XIMAG(IP);

               I2REAL = XREAL(I) - XREAL(IP);
               I2IMAG = XIMAG(I) - XIMAG(IP);

               XREAL(IP) = (I2REAL * UREAL) -
     1         (I2IMAG * UIMAG);
               XIMAG(IP) = (I2REAL * UIMAG) +
     1         (I2IMAG * UREAL);

               XREAL(I) = TREAL;
               XIMAG(I) = TIMAG;
```

```
            ENDDO;

            UREAL = (UREAL * COSX ) - (UIMAG * (-SINX));
            UIMAG = (UREAL * (-SINX) ) + (UIMAG * COSX);

          ENDDO;

      ENDDO;

      ENDDO;

STOP;
      END.
```

```
C THIS IS THE FORTRAN CHEBYSHEV COSINE ROUTINE.
C  THE COSINE OF ALL INTEGER ANGLES FROM 0 TO
C 180 DEGREES IS COMPUTED.  THE COSINE OF EACH
C ANGLE IS COMPUTED 'K' TIMES FOR TIMING
C PURPOSES.

      SUBROUTINE COSINE

      INTEGER I,J,K,L,M,N
      REAL PI,TEMP,R1,R2,R3,LIMIT,FANS

      COMMON /MCS/ I,J,K,L,M,N,PI,TEMP,R1,R2,R3,
     1        LIMIT,FANS(1:180)

      DO WHILE (J .LE. K)

          I = 0

          DO WHILE (I .LE. 180)

              IF (I .LE. 90) THEN
                      TEMP = ((I*PI)/LIMIT)
              ELSE
                      TEMP = (((N-I)*PI)/LIMIT)
              ENDIF

              FANS(I) = R1-(R2*TEMP*TEMP)+
     1                (R3*TEMP*TEMP*TEMP*TEMP)

              IF (I .GT. 90)  FANS(I) = FANS(I) * (-1)

              I = I + 1

          END DO

              I = I + 1

      END DO

      END  ! OF MYCOSINE
```

```
\ THIS SUBROUTINE IS WRITTEN IN JRS HLL AND CALCULATES THE
  COSINE OF THE ANGLES FROM L TO M DEGREES.  THE DEGREES
  ARE FIRST CONVERTED TO RADIANS AND THEN THE CHEMBYSHEV
  APPROXIMATION IS USED FIND THE VALUE OF THE COSINE.
  THE LOOP IS EXECUTED K TIMES TO ALLOW FOR TIMING OF THE
  PROCEDURE.                                              \

PROGRAM COSINE;

INTEGER I,J,K,L,M,N;
REAL PI,TEMP,R1,R2,R3,FACTOR;
REAL ARRAY ANS(180);

DO J = 1 TO K;          \ LOOP TO CONTROL THE NUMBER OF TIMES
                        THE COSINES ARE CALCULATED          \

            DO I = L TO M; \ LOOP TO CONTROL WHAT ANGLES
                           TO CALCULATE THE COSINE FOR      \

            IF (I .LE. 90) THEN

                    TEMP = (( FLOAT(I) * PI)/FACTOR)

            ELSE

                    TEMP = ((FLOAT(N-I) * PI)/FACTOR);

            ANS(I) = R1 - (R2*TEMP*TEMP) +
                     (R3*TEMP*TEMP*TEMP*TEMP);

            \ CORRECT THE SIGN OF THE ANSWER \

            IF (I .GT. 90) THEN

                    ANS(I) = (- ANS(I));

    ENDDO;

ENDDO;

STOP;
END.
```

APPENDIX C

SORTING/SEARCHING ALGORITHMS

```
C THIS SUBROUTINE LOOKS FOR EACH ITEM IN THE
C ARRAY 'KEYS' STARTING WITH THE FIRST ITEM AND
C ENDING UP WITH THE LAST ITEM.  EACH TIME AN
C ITEM IS FOUND THE INDEX OF THE DESIRED ITEM
C IS COMPARED WITH THE INDEX OF THE FOUND ITEM
C TO INSURE THAT THE CORRECT ITEM WAS FOUND.
C IF AN IMPROPER ITEM IS FOUND THEN THE COUNT
C OF ERRORS IS INCREMENTED BY ONE.
C
      SUBROUTINE BINARY SEARCH

      INTEGER KOUNT,RESULT,SIZE1,KEYS,K,UPPER,LOWER,I,J,
     1        ERRORS,F

      COMMON /NOS/ KEYS(10000),KOUNT,RESULT, SIZE1,UPPER,
     1        LOWER,I,J,F,ERRORS,K

C LOOP THROUGH EVER ELEMENT OF THE ARRAY
C AND LOOK FOR EACH ELEMENT ONCE.

      DO J = 1,SIZE1

C INITIALIZE THE CONSTANTS AND VARIABLES

                KOUNT = 0
                RESULT = KEYS(J)
                UPPER = SIZE1
                LOWER = 1
                F = .FALSE.

                IF ( RESULT .LT. KEYS(LOWER) ) THEN
                        RETURN
                ELSE IF ( RESULT .GT. KEYS(UPPER) ) THEN
                        RETURN
                  ELSE
                  DO WHILE (F .EQ. .FALSE. )
                     I = ( UPPER + LOWER + 1)/2
                     IF( RESULT .LT. KEYS(I) ) THEN
                                UPPER = I-1
                     ELSE IF ( RESULT .GT. KEYS(I) ) THEN
                                LOWER = I+1
                          ELSE IF ( RESULT .EQ. KEYS(I) ) THEN
                                     F = .TRUE.
                               ELSE
                                    RESULT = -3
                                    F = .TRUE.
```

```
                    ENDIF
                IF ( UPPER .LT. LOWER ) THEN
                    F = .TRUE.
                ELSE
                    KOUNT = KOUNT+1
                ENDIF
              END DO
            ENDIF

            IF (I .NE. J) ERRORS = ERRORS + 1

END DO

END ! OF BINARY SEARCH SUBROUTINE
```

```
\ BINARY SEARCH PROGRAM WRITTEN IN JRS HLL \
\ WHEN THE RESULT IS ASSIGNED A NEGATIVE   \
\ VALUE, THERE IS AN ERROR IN THE RESULTS  \

     PROGRAM BSEARCH;
     INTEGER ARRAY KEYS(10000);
     INTEGER KOUNT,RESULT,SIZE1,UPPER,LOWER,
             I,J,FLAG,ERRORS,K;

     DO K = 1 TO SIZE1;

             KOUNT = 0;
             RESULT = KEYS(K);
             UPPER = SIZE1;
             LOWER = 1;
             FLAG = 0;
             IF ( RESULT .LT. KEYS(LOWER) ) THEN
                  RESULT= -1
             ELSE
                  IF ( RESULT .GT. KEYS(UPPER) ) THEN
                       RESULT= -2
                  ELSE
                       DO WHILE (FLAG .EQ. 0);
                         I = ( UPPER + LOWER + 1)/2;
                         IF( RESULT .LT. KEYS(I) ) THEN
                                UPPER = I-1
                         ELSE
                            IF (RESULT .GT. KEYS(I)) THEN
                                LOWER = I+1
                            ELSE
                              IF (RESULT .EQ. KEYS(I)) THEN
                                 FLAG = 1
                              ELSE
                                DO;
                                   RESULT = -3;
                                   FLAG =1;
                                ENDDO;
                         IF ( UPPER .LT. LOWER ) THEN
                                FLAG = 1
                         ELSE
                                KOUNT = KOUNT+1;
                       ENDDO;

             IF (K .NE. I) THEN
                       ERRORS = ERRORS + 1;

     ENDDO;

        STOP;
     END.
```

```
C THIS IS THE QUICK SORT ALGORITHM IN FORTRAN
C 'A' IS THE ARRAY HOLDING THE ITEMS TO BE SORTED
C ALL INTEGERS IN 'A' ARE GENERATED BY THE HARNESS
C PROGRAM
C
      SUBROUTINE SORT

      INTEGER I,M,J,P,T,Q,K,Q1,X,N,LT,UT,A
      COMMON /NCS/ I,M,J,P,T,Q,K,Q1,X,N,LT(14),UT(14),
     1        A(50000)

C INITIALIZE THE VARIABLES AND CONSTANTS

      J = N
      I = 1
      M = 1

200   IF (J - I .GT. 1) THEN
         P = (J + I)/2
         T = A(P)
         A(P) = A(I)
         Q = J
         DO 300 K = I + 1,Q
            IF (A(K) .GT. T) THEN
               DO 201 Q1 = Q,K,-1
                  IF (A(Q1) .LT. T) THEN
                     X = A(K)
                     A(K) = A(Q1)
                     A(Q1) = X
                     Q = Q1 - 1
                     GOTO 120
                  ENDIF
201            CONTINUE
               Q = K - 1
               GOTO 140
120         ENDIF
300      CONTINUE

140      A(I) = A(Q)
         A(Q) = T
         IF (2*Q .GT. I+J) THEN
            LT(M) = I
            UT(M) = Q - 1
            I = Q + 1
         ELSE
            LT(M) = Q + 1
            UT(M) = J
            J = Q - 1
         ENDIF

         M = M + 1
```

```
            GOTO 200
         ELSE
            IF (I .GE. J) THEN
               GOTO 160
            ELSE
               IF (A(I) .GT. A(J)) THEN
                  X = A(I)
                  A(I) = A(J)
                  A(J) = X
               ENDIF
160            M = M - 1
               IF (M .GT. 0) THEN
                  I = LT(M)
                  J = JT(M)
                  GOTO 200
               ENDIF
            ENDIF
         ENDIF

         END    ! OF SORT
```

```
PROGRAM SORT;

\ THIS PROGRAM SORTS THE ELEMENTS OF AN ARRAY INTO
  ASCENDING ORDER.  THE METHOD USED IS THE "QUICKERSORT"
  ALGORITHM OF R.S. SCOWEN, ALGORITHM  271, CACM, VOL.
  8, NUMBER 11, OCTOBER 1965.  THIS VERSION WAS COPIED
  FROM THE JRS HLL MANUAL FOR THE AMGS SYSTEM.

  THE ALGORITHM WORKS BY CONTINUALLY SPLITTING THE ARRAY
  INTO PARTS SUCH THAT ALL ELEMENTS OF ONE PART ARE LESS
  THAN ALL ELEMENTS OF THE OTHER, WITH A THIRD PART
  IN THE MIDDLE CONSISTING OF A SINGLE ELEMENT.

  THE ARRAY TO BE SORTED IS PRE-SET IN 'A' AND THE NUMBER
  OF ELEMENTS IN THE ARRAY IS SET IN 'N'.  ON EXIT, THE
  ELEMENTS OF ARRAY 'A' ARE SORTED.                    \


INTEGER I,M,J,P,T,Q,K,Q1,X,N;

INTEGER ARRAY LT(14),UT(14),A(50000);


      J = N;
      I = 1;
      M = 1;

100:  IF (J-I.GT.1) THEN
        DO;
          P = (J+I)/2;
          T = A(P);
          A(P) = A(I);
          Q = J;
          DO K=I+1 TO J;
            IF (A(K).GT.T) THEN
              DO;
                DO Q1=N DOWNTO K;
                  IF (A(Q1).LT.T) THEN
                    DO;
                      X = A(K);
                      A(K) = A(Q1);
                      A(Q1) = X;
                      Q = Q1-1;
                      GOTO 120;
                    ENDDO;
                ENDDO;
              Q = K-1;
              GOTO 140;
            ENDDO;
120:        CONTINUE;
        ENDDO;
```

```
140:      A(I) = A(Q);
          A(Q) = T;
          IF (2*Q .GT. I+J) THEN
            DO;
              LT(M) = I;
              UT(M) = Q-1;
              I = Q+1;
            ENDDO
          ELSE
            DO;
              LT(M) = Q+1;
              UT(M) = J;
              J = Q-1;
            ENDDO;
          M = M+1;
          GOTO 100;
        ENDDO

      ELSE IF (I.GE.J) THEN GOTO 160
      ELSE
        DO;
          IF (A(I).GT.A(J)) THEN
            DO;
              X    = A(I);
              A(I) = A(J);
              A(J) = X;
            ENDDO;

160:      M = M-1;
          IF (M.GT.0) THEN
            DO;
              I = LT(M);
              J = UT(M);
              GOTO 100;
            ENDDO;
        ENDDO;

STOP;
END.
```

```
C SIEVE PROGRAM IN FORTRAN IV
C COPIED FROM BYTE MAGAZINE, JAN 83
C THE SIEVE OF ERATOSTHENES ALGORITHM IDENTIFIES
C THE PRIME NUMBERS FROM 3 TO N.  IN THIS CASE
C N = 16,381.  THE PRIMES ARE STORED IN
C AN ARRAY NAMED 'PRIMES' FOR VERIFICATION
C OF THE ALGORITHM IN THE HARNESS PROGRAM

      SUBROUTINE SIEVESUB

      INTEGER I,J,K,COUNT,ITER,PRIME,N,PRIMES
      LOGICAL FLAGS

      COMMON /F/ FLAGS(8191)
      COMMON /STORE/ I,J,K,COUNT,ITER,PRIME,N,PRIMES(1900)

      DO 192 ITER = 1,20
         COUNT = 0
         N = 1
         DO 110 I = 1,8191
110         FLAGS(I) = .TRUE.
         DO 191 I = 1,8191
            IF (.NOT. FLAGS(I)) GOTO 191
            PRIME = I + I + 1
            PRIMES(N) = PRIME
            N = N + 1
            COUNT = COUNT + 1
            K = I + PRIME
            IF (K .GT. 8191) GOTO 191
            DO 160 J = K, 8191, PRIME
160            FLAGS(J) = .FALSE.
191      CONTINUE
192   CONTINUE

      END  ! OF SIEVESUB
```

```
\ ALL VERSION OF THE SIEVE OF ERATOSTHENES
  THE PROGRAM IDENTIFIES THE PRIME NUMBERS BETWEEN
  1 AND N.                                          \

PROGRAM SIEVE;


INTEGER  I,J,K,COUNT,L,PRIME,ZERO,M,TEN;

INTEGER ARRAY FLAGS(8191), PRIMES(1900);

     DO L = 1 TO TEN;

        COUNT = 0;
        J = 1;

        DO I = 1 TO M;
           FLAGS(I) = 1;
        ENDDO;

        DO I = 1 TO M;
           IF (FLAGS(I) .EQ. 1) THEN
           DO;
              PRIME = I + I + 1;
              PRIMES(J) = PRIME;
              J = J + 1;
              COUNT = COUNT + 1;
              K = I + PRIME;

              DO WHILE (K .LE. M);
                 FLAGS(K) = 0;
                 K = K + PRIME;
              ENDDO;

           ENDDO;

        ENDDO;

     ENDDO;

STOP;

END.
```

```
C BUBBLE SORT IN FORTRAN
C THE INTEGERS IN ARRAY 'A' ARE SORTED INTO
C ASCENDING ORDER BY CONTINUALLY MOVING THE
C 'NEXT' LARGEST ITEM TO ITS PROPER POSITION
C IN THE ORDERING.  THE ALGORITHM IS IMPROVED
C BY CHECKING EACH TIME THROUGH THE SORT TO
C SEE IF ANY EXCHANGES HAVE BEEN MADE.  IF
C NONE ARE MADE THEN THE PROGRAM TERMINATES.

      SUBROUTINE BUBBLE

      INTEGER I,N,XCHANG,TEMP,A
      COMMON /ACS/ I,N,XCHANG,TEMP,A(10000)

      XCHANG = .TRUE.

      DO WHILE(XCHANG .EQ. .TRUE.)

               XCHANG = .FALSE.
               N = N - 1
               DO I = 1,N
                        IF (A(I) .GT. A(I+1)) THEN
                                 TEMP = A(I)
                                 A(I) = A(I+1)
                                 A(I+1) = TEMP
                                 XCHANG = .TRUE.
                        ENDIF
               END DO
      END DO

      END  ! OF BUBBLE
```

```
\ THIS IS THE BUBBLE SORT IN JRS HLL
  ARRAY 'A' HOLDS THE INTEGERS TO BE SORTED. \

PROGRAM BUBL;

INTEGER I,N,XCHANG,TEMP;
INTEGER ARRAY A(10000);

XCHANG = 1;

DO WHILE (XCHANG.NE.0);
   N = N-1;
   XCHANG = 0;

   DO I=1 TO N;
      IF (A(I).GT.A(I+1)) THEN
       DO;
         TEMP = A(I);
         A(I) = A(I+1);
         A(I+1) = TEMP;
         XCHANG = 1;
      ENDDO;

   ENDDO;

ENDDO;
STOP;
END.
```

APPENDIX D

BIT MANIPULATION ALGORITHMS

```
C BIT MANIPULATION PROGRAM IN FORTRAN
C ARRAY 'A' HOLDS THE PREGENERATED VALUES TO
C BE MANIPULATED.  'N' HOLDS THE NUMBER OF
C TIMES THE MANIPULATION WILL OCCUR FOR TIMING
C PURPOSES.

      SUBROUTINE BITMANIP

      INTEGER I,N,ROT,A

      COMMON /NCS/ I,N,ROT,A(100000)

      DO 400 I = 1,N

               A(I) = IIAND(A(I),A(I))

               A(I) = JNOT(A(I))

               A(I) = JNOT(A(I))

               A(I) = JIABS(A(I))

               A(I) = JIBITS(A(I),0,32)

               A(I) = IAND(A(I),A(I))

               A(I) = IOR(A(I),A(I))

               A(I) = JISHFTC(A(I),32,32)

400   END DO

      END    ! OF BITMANIP
```

```
\ BITMANIPULATION PROGRAM IN JRS HLL.  ARRAY 'A'
  HOLDS THE VALUES TO BE MANIPULATED.  N IS THE
  TOTAL NUMBER OF TIMES THE ITEMS WILL BE
  MANIPULATED.  'I' IS THE LOOPING VARIABLE.   \

    PROGRAM BITMANIP;

    INTEGER I,N,ROT;
    INTEGER ARRAY A(100000);

    DO I = 1 TO N;

            A(I) = A(I) .AND. A(I);

            A(I) = A(I) .XOR. (MASK(31,0));

            A(I) = A(I) .XOR. (MASK(31,0));

            A(I) = ABS(A(I));

            A(I) = SRL((A(I) .AND. (MASK(31,0))),0);

            A(I) = A(I) .AND. A(I);

            A(I) = A(I) .OR. A(I);

            A(I) = RLL(A(I),32);

    ENDDO;

    STOP;

    END.
```

```
      SUBROUTINE BITREV
C
C     ***************************************************
C
C     X      -  COMPLEX ARRAY X(2**M)
C
C     M      -  NUMBER OF POINTS
C
C        BASED UPON AN FFT FIRST DEVELOPED BY SIGNALS
C        SCIENCE CORPORATION FOR PROJECT SALESCLERK.
C
C        FIRST TRANSCRIBED BY LCDR C LAURVICK, USN
C        MODIFIED BY LT M HARTONG, USN
C
C     ***************************************************
C
         COMPLEX X(4096),T
         INTEGER N,NV2,NM1,M,J,K
C
C     REARRANGE ARRAY- BIT REVERSAL
C
         N=2**M
        NV2=N/2
        NM1=N-1
        J=1
        DO 30 I=1,NM1
                IF(I.GE.J) GO TO 25
                T=X(J)
                X(J)=X(I)
                X(I)=T
25              K=NV2
26              IF(K.GE.J) GO TO 30
                J=J-K
                K=K/2
                GO TO 26
30      J=J+K
        RETURN
        END
```

```
    PROGRAM BITREV;

\ ************************************************
  BIT REVERSAL FOR FFT - AMGS HLL VERSION
  ************************************************

  BASE UPON AN FFT FIRST DEVELOPED BY SIGNALS
  SCIENCE CORP.  FOR PROJECT SALESCLERK.
  TRANSLATED INTO HLL FOR THE NCS BY LT M HARTONG
  ************************************************\

    INTEGER I,J,M,N,L,LE0,LE1,IP,KOUNT,K,
              NV2,NM,NM1,P;

    REAL UREAL,UIMAG,PHASE,COSX,SINX,TREAL,TIMAG,
           T2REAL,T2IMAG,TMP,PI,A1,A2,A3,K3;

    REAL ARRAY XREAL(4096),XIMAG(4096);

    \ REPEAT 30 TIMES FOR TIMING PURPOSES  \

    DO L = 1 TO 30;

    \ N = 2**M                                \

          N = 1;

          DO KOUNT = 1 TO M;
                N = N * 2;
          ENDDO;

    \ INITIALIZE THE CONSTANTS                \

          NV2 = N/2;
          NM1 = N - 1;
          J = 1;

    \       REARRANGE ARRAY- BIT REVERSAL   \

          DO  I= 1 TO NM1;
            IF (I.GE.J) THEN GOTO 25;
              TREAL = XREAL(J);
              TIMAG = XIMAG(J);
              XREAL(J) = XREAL(I);
              XIMAG(J) = XIMAG(I);
              XREAL(I) = TREAL;
              XIMAG(I) = TIMAG;
           25:K=NV2;
           26:IF (K.GE.J) THEN GOTO 30;
              J=J-K;
              K=K/2;
```

```
                GOTO 26;
            30:J=J+K;
          ENDDO;

ENDDO;       \ END OF LOOP L  \

STOP;

END.
```

APPENDIX E

SAMPLE HARNESS SETUP

```
      PROGRAM FACTORIAL

C THIS IS THE FACTORIAL PROGRAM, FORTRAN VERSION
C AN INTEGER IS READ AND THE FACTORIAL OF THAT
C INTEGER IS PRINTED.  THE FACTORIAL OF THAT
C NUMBER IS CALCULATED 100,000 TIMES BEFORE BEING
C PRINTED FOR TIMING PURPOSES.

      INTEGER TOTAL, VALUE, TEMP, COUNT, ZERO,
     1        TIMER, HANDLE, IRET, TMST
      INTEGER TIMES, C, V, T, ANS, ONE

      COMMON /NOS/ TOTAL, VALUE, TEMP, COUNT, ZERO, ONE

10    FORMAT(/,' ENTER AN INTEGER BETWEEN 1 AND 12',/)
20    FORMAT(' THE FACTORIAL OF ',I2, ' IS ', I9)
30    FORMAT(/,' FACTORIAL USING FORTRAN SUBROUTINE WITH
     1        COMMON'/)
40    FORMAT(/,' FACTORIAL USING IRS HLL WITH COMMON'/)
50    FORMAT(/,' FACTORIAL USING STRAIGHT FORTRAN CODE
     1        WITH COMMON'/)
60    FORMAT(/,' CPU TIME = ',F5.2,' SECONDS'/)
70    FORMAT(I2)
80    FORMAT(/,' FACTORIAL USING STRAIGHT FORTRAN
     1        WITHOUT COMMON'/)

C READ WHAT FACTORIAL TO DETERMINE

      WRITE(6,10)
      READ(5,70) TEMP

C INITIALIZE THE CONSTANTS

      TIMES = 100000
      COUNT = TIMES ! NUMBER OF TIMES TO EXECUTE LOOP
      ZERO = 0
      TOTAL = 1
      ONE = 1
      C = TIMES
      T = TEMP


C THIS PART IS STRAIGHT FORTRAN WITHOUT COMMON

      WRITE(6,80)
```

```
      IF (.NOT. LIB$INITTIMER(HANDLE)) CALL ERR

      DO WHILE (C .GT. 0)

              C = C - 1

              V = I

              ANS = 1

              DO WHILE (V .GT. 0)

                      ANS = ANS * V

                      V = V - 1

              END DO

      END DO

      IF (.NOT. LIB$STATTIMER(2,TIMER,HANDLE)) CALL ERR

      WRITE(6,20) I, ANS
      WRITE(6,60) FLOAT(TIMER)/100.0

C THIS IS THE STRAIGHT FORTRAN CODE VERSION WITH COMMON

      WRITE(6,50)

      COUNT = TIMES
      TOTAL = ONE

      IF (.NOT. LIB$INITTIMER(HANDLE)) CALL ERR

      DO WHILE (COUNT .GT. 0)
          COUNT = COUNT - 1
          VALUE = TEMP
          TOTAL = ONE

          DO WHILE (VALUE .GT. 0)
              TOTAL = TOTAL * VALUE
              VALUE = VALUE - 1
          END DO
      END DO

      IF (.NOT. LIB$STATTIMER(2,TIMER,HANDLE)) CALL ERR

      WRITE(6,20) TEMP, TOTAL
      WRITE(6,60) FLOAT(TIMER)/100.0
```

```
C THIS PART IS A SUBROUTINE CALL IN FORTRAN WITH COMMON

      COUNT = TIMES
      TOTAL = ONE

      WRITE(6,30)

      IF (.NOT. LIB$INITTIMER(HANDLE)) CALL ERR

               CALL FAC

      IF (.NOT. LIB$STATTIMER(2,TIMER,HANDLE)) CALL ERR

      WRITE(6,20) TEMP, TOTAL
      WRITE(6,60) FLOAT(TIMER)/100.0


C THIS PART USES JRS HLL WITH COMMON

      TOTAL = ONE
      COUNT = TIMES

      WRITE(6,40)

      IF (.NOT. LIB$INITTIMER(HANDLE)) CALL ERR

         CALL XFCC(TOTAL, IRET, INST)

      IF (.NOT. LIB$STATTIMER(2,TIMER,HANDLE)) CALL ERR

      WRITE(6,20) TEMP, TOTAL
      WRITE(6,60) FLOAT(TIMER)/100.0

      END


C THE FACTORIAL SUBROUTINE IN FORTRAN

      SUBROUTINE FAC

      INTEGER TOTAL, VALUE, TEMP, COUNT, ZERO, ONE
      COMMON /VCS/ TOTAL, VALUE, TEMP, COUNT, ZERO, ONE

      DO WHILE (COUNT .GT. ZERO)

         COUNT = COUNT - 1
         VALUE = TEMP
         TOTAL = ONE

         DO WHILE (VALUE .GT. ZERO)
```

```
        TOTAL = TOTAL * VALUE
        VALUE = VALUE - 1

    END DO

END DO

END ! OF SUMM
```

```
\ FACTORIAL PROGRAM IN JRS HLL \

     PROGRAM FACTORIAL;

     INTEGER TOTAL, VALUE, TEMP, COUNT, ZERO, I;


     DO WHILE (COUNT .GT. ZERO);

        COUNT = COUNT - 1;
        VALUE = TEMP;
        TOTAL = 1;

        DO WHILE (VALUE .GT. ZERO);
           TOTAL = TOTAL * VALUE;
           VALUE = VALUE - 1;
        ENDDO;

     ENDDO;

     STOP;

     END.
```

```
C SUBROUTINE ERR IS USED FOR SIGNALING ERRORS FROM
C THE TIMING MECHANISM.

      SUBROUTINE ERR

          WRITE(6,102)
102       FORMAT (' PROBLEM WITH THE LIBRARY CALL')

      END    ! OF ERR
```

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	LtCol Alan A. Ross Code 52Rs Naval Postgraduate School Monterey, California 93943-5100	2
4.	Mr. Andrew J. Fox Code 7752 U. S. Naval Research Laboratory 4555 Overlook Ave. S.W. Washington, D.C. 20375	1
5.	Mr. Erwin H. Warshawsky 202 W. Lincoln Ave. Orange, California 92665	1
6.	Capt Terry J. Newton 1734 Alexander Circle Pueblo, Colorado 81001	2
7.	Lt Mark Hartong P. O. Box 3249 Vallejo, California 94590-9998	1
8.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
9.	Professor H. H. Loomis Code 62Lm Naval Postgraduate School Monterey, California 93943-5100	1

10. Computer Technology Programs 1
Code 37
Naval Postgraduate School
Monterey, California 93943-5100

214751

Thesis
N46943
c.1

Newton
Performance evaluation of the JRS automatic microcode generating system.